GeoAR Android App and Backend POI Helper

Aishna Agrawal, Aakash Thakkar, Alex Pang

University of California, Santa Cruz

Abstract

Augmented Reality (AR) is an enhanced version of reality where live direct or indirect views of physical real-world environments are augmented with superimposed computer-generated images over a user's view of the real-world, thus enhancing one's current perception of reality. In this project, we experiment with Markerless AR which relies on location data instead of physical markers. Points of interest (POI) are saved in the backend as latitude and longitude [1] information that are queried by the front-end Android application. The app integrates Google's Augmented Reality SDK called ARCore [2] to help render objects at various points of interest in the real world.

To provide significant value to a users life, any geo-location based application must be able to model the real world objects and provide unique dynamic data about them in a creative manner. They must also have a well defined API to remain platform independent. We try to solve this problem with the use of POI Helper which creates a backend as a scalable API service. This project strives to make it easy for developers to make a wide variety of AR applications by abstracting the commonly performed tasks and confining them to a basic structure without constraining the developer's ability to customize their product. This report describes the development of an open-source Android application, general purpose back-end server and a scalable web-application to visualize the POIs. To demonstrate the usage of the app, a sample app on a UCSC tour was implemented.

Keywords: Augmented Reality, Markerless, Geo-location

1. Introduction

1.1. Motivation

Both video games and cell phones are driving the development of augmented reality. Everyone from tourists, to soldiers, to someone looking for the closest subway stop can now benefit from the ability to place computer-generated graphics in their field of vision. Both Apple and Google, have recently released augmented reality platforms called ARKit and ARCore respectively. Since it's a relatively new technology and has immense potential, the idea was to experiment with one of these and create a framework that can be made open-source to make it easier to develop AR apps. As the structure of any location based AR app has to be similar this project aims to be the go to plug and play AR app generator.

1.2. Scope

The application serves as a base for any developer to build their own Geolocation AR app. The basic concept of the app is to help a user identify a certain Point of Interest(POI). When the user points their camera at a building, a 3D object like a wooden sign representing the direction and name of the building, is augmented onto their camera view. The user can view this object from any angle, interact with it by touching it for further information about the point of interest. The sample app developed as proof of concept is a UCSC tour app that helps new UCSC students identify cafes, libraries, parking spots and academic building all around campus.

1.3. Report structure

This report aims to help the developer understand each building block thoroughly with description of each component as well as code documentation. In the sections below, we will discuss background and related work, approach to the front-end app development, back-end development, followed by results, conclusion and expected future work.

2. Background

Though it's been in development since the early 2000s, Augmented Reality did not get popular until last year's release of Pokemon Go. Until recently, real AR applications required the use of a tracking image such as a QR code or image marker to function. The marker would be read and interpreted by camera-equipped hardware capable of running specialized software. The smartphone, translates that information into a 3D model or animation that maintains a consistent position within the scene, regardless of how or where the user moves.Though it works well, marker-based AR can be limiting and inconvenient. Markerless Augmented Reality is now ready for mainstream adoption, thanks to Apple's ARKit and Google's ARCore that were released this year.

There are a lot of AR SDKs like Wikitude, Vuforia that integrate location to create Geo-location AR apps but all of these AR engines are licensed and do not integrate well with native app development. Also, these proposals neither provide insights into the functionality of such an engine nor its customization to a specific purpose. As ARCore is available for all users to view and use freely, projects developed with it can be made open-source. Since it does not support location-based AR in its preview version, this project is an experiment to integrate it with Android's Location and Sensor data to create a location supporting version of Google ARCore.
Every location based application currently being developed or that is available to download holds some form of back end data store that feeds the app point of interest data to be rendered onto the user's view. With the growth of Augmented Reality, and Geo-Location based mobile applications it is redundant for every organization to develop and update their own back end service. It is in the benefit of the community to have a generic, open source back end capable of supporting diverse application so that developers can focus solely on creating better quality applications at low cost. The most popular back end with a pre-built data store is provided by Google as a part of their Google Places API. But Google Places API is a commercial product that constraints the developers to the data provided by Google, they are charged per request and the data structure is not customizable to power a wide variety of applications.

There exist several other backend as a service platforms most popular one being FireBase, AWS Lambda etc. These are focused more towards real time event driven fixed structure data, and not designed to hold a wide variety of flexible data options.

This project is designed to experiment with various standard technology stacks and technologies used in creating backends, evaluate the differences among them, picking the best stack for the task and to form the base for an open source project to creating a simple method of generating augmented reality applications for a wide variety of purposes without spending considerable time on setting up the platform.

2.1. Related work

In (Geiger 2014) [3], the authors discussed how to develop the core of a location-based augmented reality engine for Android and iOS mobile phones. The issue with this report is that it is only designed to render 2d points at POI and not 3d objects that creates augmented reality in its true essence. Also, this framework was created before Google and Apple released AR engines compatible with their native development.

There are a few location-based AR apps that are already in the market that were built using existing AR SDKs or their own AR engines. For example, Yelp Monocle [4], the monocle is a hidden feature on the Yelp app which opens up some cool features for users who discover it. If you point your camera, you'll see boxes pop up for nearby businesses or services in the direction of where you are pointing at. You can filter to see just restaurants, just bars, places your friends have been, or everything.

3. Android application

The application relies on GPS and built-in sensors to determine location and orientation of the device. When the user opens the app, camera view is displayed. Using the GPS, current location of the user is determined and if the user is within the geographic region that the app targets, an API call is made to the server to fetch markers that will be appropriately displayed. When the user points their camera at a certain location that has a corresponding virtual marker based on calculations using current location and sensor data, a 3D object is rendered on the camera view thereby augmenting the user's reality by meshing the physical and virtual world. The location is represented by a latitude and longitude, for larger areas such as trails or paths, a set of virtual markers can be organized. On tapping the object, information about the location will be displayed.

3.1. Software and hardware requirements

The code is written entirely in Java for Android, available on GitHub: `https://github.com/aishnacodes/Geolocation-ARCore`. Requirements to set up the project are as following:

- IDE - Android Studio 3.0
- Mobile phone - Currently works on Samsung S8, Google Pixel, Pixel XL, Pixel 2, Pixel 2 XL (as of Dec 2017)

3.2. Tools and components explained

The app involves AR, sensors, GPS, Computer Graphics and server communication. Each component has extensive libraries [5] provided by Android, the following section explains which libraries were used and how they are integrated within the app.

3.2.1. Google ARcore

ARCore [6] is a platform for building augmented reality apps on Android. It uses Motion Tracking and Light Estimation to integrate virtual content with the real world as seen through your phone's camera. Motion tracking allows the phone to understand and track its position relative to the world. As your phone moves through the world, ARCore uses a process called concurrent odometry and mapping(COM), to understand where the phone is relative to the world around it. ARCore detects visually distinct features in the captured camera image called feature points and uses these points to compute its change in location. The visual information is combined with inertial measurements from the device's Inertial Measurement Unit(IMU) to estimate the pose (position and orientation) of the camera relative to the world over time. Light estimation allows the phone to estimate the environment's current lighting conditions.

(a) **Config** (public final class Config)
Holds settings that are used to configure the session.

- public static Config createDefaultConfig()

  Returns a sensible default configuration. Plane detection and lighting estimation are enabled, and blocking update is selected. This configuration is guaranteed to be supported on all devices that support ARCore.

  ```
      mDefaultConfig = Config.createDefaultConfig();
  ```

(b) **Frame** (public final class Frame)
Provides a snapshot of AR state at a given timestamp.

(i) public Frame.TrackingState getTrackingState()
Returns the state of the AR tracking system.

```
if (frame.getTrackingState() == TrackingState.NOT_TRACKING) {
    return;
    }
```

(ii) public Pose getPose()
Returns the pose of the user's device in the world coordinate frame at the time of capture of the current camera texture. The position of the pose is located at the device's camera. The orientation of the pose matches the orientation of the display (considering display rotation) and uses OpenGL conventions (+X right, +Y up, -Z in the direction the camera looks).

```
    Pose pose = frame.getPose();
```

(iii) public LightEstimate getLightEstimate()
Returns the current ambient light estimate. Returns the pixel intensity of the current camera view. Values are on the range (0.0, 1.0), with zero being black and one being white.

```
frame.getLightEstimate().getPixelIntensity();
```

(c) **Pose** (public final class Pose)
Represents an immutable rigid transformation from one coordinate frame to another. As provided from all ARCore APIs, Poses always describe the transformation from object's local coordinate frame to the world coordinate frame. That is, Poses from ARCore APIs can be thought of as equivalent to OpenGL model matrices. The transformation is defined using a quaternion rotation about the origin followed by a translation.

public Pose (float[] translation, float[] rotation)

- **Translation**
  Translation is the position vector from the destination (usually world) coordinate frame to the local coordinate frame, expressed in destination (world) coordinates.
- **Rotation**
  Rotation is a quaternion following the Hamilton convention. Assume the destination and local coordinate frames are initially aligned, and the local coordinate frame is then rotated counter-clockwise about a unit-length axis, k, by an angle, theta. The quaternion parameters are hence:

$$x = k.x * sin(\theta/2)$$

$$y = k.y * sin(\theta/2)$$

$$z = k.z * sin(\theta/2)$$

$$w = cos(\theta/2)$$

(d) **Anchor** (public final class Anchor)
Describes a fixed location and orientation in the real world. To stay at a fixed location in physical space, the numerical description of this position will update as ARCore's understanding of the space improves. Use getPose() to get the current numerical location of this anchor. This location may change any time update() is called, but will never spontaneously change.

- Pose getPose()

  Returns the pose of the anchor in the world coordinate frame. This pose may change each time update() is called. This pose should only be considered valid if getTrackingState() returns TRACKING.

```
Pose pose = anchor.getPose();
```

(e) **Session** (public final class Session)
Manages AR system state and handles the session lifecycle. This class is the main entry point to ARCore API. This class allows the user to create a session, configure it, start/stop it, and most importantly receive frames that allow access to camera image and device pose.

(i) public void getProjectionMatrix (float[] dest, int offset, float near, float far)
Returns a projection matrix for rendering virtual content on top of the camera image.

```
mSession.getProjectionMatrix(projmtx, 0, 0.1f, 100.0f);
```

(ii) Frame update()
Updates the state of the ARCore system. This includes: receiving a new camera frame, updating the location of the device, updating the location of tracking anchors

```
Frame frame = mSession.update();
```

(iii) Anchor addAnchor (Pose pose)
Defines a tracked location in the physical world.

```
Anchor anchor = mSession.addAnchor(frame.getPose());
```

3.2.2. Geolocation

Geolocation requires an internet connection and GPS-enabled mobile phone. The application accesses the network provider's internet and GPS to identify the user's current location. Location managers and listeners are used to constantly update the user's position and re-calculate its relative position with markers.

(a) **LocationManager** (android.location.LocationManager)
This class provides access to the system location services. These services allow applications to obtain periodic updates of the device's geographical location

```
this.mLocationManager = (LocationManager)
     this.getSystemService(this.LOCATION_SERVICE);
```

(i) getLastKnownLocation()
Returns a Location indicating the data from the last known location fix obtained from the given provider.

```
mLocationManager.getLastKnownLocation
    (LocationManager.NETWORK_PROVIDER);
```

(ii) requestLocationUpdates()
Register for location updates using the named provider, and a pending intent.

```
mLocationManager.requestLocationUpdates(
    LocationManager.NETWORK_PROVIDER,
    MIN_TIME_BW_UPDATES,
    MIN_DISTANCE_CHANGE_FOR_UPDATES, this
    );
```

(b) **LocationListener** (android.location.LocationListener)
Interface LocationListener is extended by the ARActivity and is used for receiving notifications from the LocationManager when the location has changed. These methods are called if the LocationListener has been registered with the location manager service using the requestLocationUpdates.

- onLocationChanged()
  Called when the location has changed. For each marker, we calculate the distance between the current location and the marker and update it in the 'distance' variable of a MarkerInfo Object using pre-defined function distanceTo() of the Location class.

```
public void onLocationChanged(Location location) {
  mLocation = location;
  MarkerInfo marker;

  for (int i = 0; i < mMarkerList.size(); i++) {
      marker = mMarkerList.get(i);
      marker.setDistance(location.
              distanceTo(marker.getLocation()));
      }
  }
```

3.2.3. Sensors

The application makes use of the mobile phone's built-in sensors that measure motion and orientation [7]. These sensors are capable of providing raw data with high precision and accuracy, and are useful for monitoring three-dimensional device movement or positioning. To be able to identify if the camera is pointing towards a marker location, the direction of the phone and orientation is required. This data is constantly gathered from the motion sensors.

(a) **Sensor Event** (android.hardware.SensorEvent)
This class represents a Sensor event and holds information such as the sensor's type, the time-stamp, accuracy and of course the sensor's data.

```
if (sensorEvent.sensor.getType() == Sensor.TYPE_ROTATION_VECTOR) {
        //calculation
        }
```

The rotation vector represents the orientation of the device as a combination of an angle and an axis, in which the device has rotated through an angle θ around an axis (x, y, z). The reference coordinate system is defined as a direct orthonormal basis, where:

- X is defined as the vector product Y.Z (It is tangential to the ground at the device's current location and roughly points East).
- Y is tangential to the ground at the device's current location and points towards magnetic north.
- Z points towards the sky and is perpendicular to the ground.

(b) **SensorManager** (android.hardware.SensorManager)
SensorManager lets you access the device's sensors.

```
mSensorManager = (SensorManager)
    this.getSystemService(SENSOR_SERVICE);
```

(i) getRotationMatrixFromVector (float[] R, float[] rotationVector)
Helper function to convert a rotation vector to a rotation matrix.

```
SensorManager.getRotationMatrixFromVector
    (rotationMatrixFromVector, sensorEvent.values);
```

(ii) remapCoordinateSystem (float[] inR, int X, int Y, float[] outR)
Rotates the supplied rotation matrix so it is expressed in a different coordinate system.
The phone is assumed to be horizontal as if it were lying on a table with its screen facing upward and the top of the phone pointing away from the user. Since we are using the camera for our augmented reality application, the phone needs to be upright, for that we need to change our coordinate system.

```
SensorManager
    .remapCoordinateSystem(rotationMatrixFromVector,
        SensorManager.AXIS_X, SensorManager.AXIS_Y,
                             updatedRotationMatrix);
```

(iii) getOrientation (float[] R, float[] values)
Computes the device's orientation based on the rotation matrix.

```
SensorManager.getOrientation
    (updatedRotationMatrix, orientationValues);
```

When it returns, the array values are as follows:

- **Azimuth**(values[0]): angle of rotation about the -z axis.
  This value represents the angle between the device's y axis and the magnetic north pole. When facing north, this angle is 0, when facing south, this angle is π.
- **Pitch**(values[1]): angle of rotation about the x axis.
  This value represents the angle between a plane parallel to the device's screen and a plane parallel to the ground. Assuming that the bottom edge of the device faces the user and that the screen is face-up, tilting the top edge of the device toward the ground creates a positive pitch angle. The range of values is $-\pi$ to π.
- **Roll**(values[2]): angle of rotation about the y axis.
  This value represents the angle between a plane perpendicular to the device's screen and a plane perpendicular to the ground. Assuming that the bottom edge of the device faces the user and that the screen is face-up, tilting the left edge of the device toward the ground creates a positive roll angle. The range of values is $-\pi/2$ to $\pi/2$.

(c) **SensorEventListener** (android.hardware.SensorEventListener)
SensorManager lets you access the device's sensors. onSensorChanged(SensorEvent event) is called every time the rotation vector event is changed. Here we check if a marker is in range of the camera view. We do this by calculating the bearing from user's current location to the marker's location using latitude and longitude for locations. The bearing is the angle that is between 2 locations with respect to magnetic north. If the Azimuth (phone's direction with respect to north) is equal to the bearing then the marker is in range. To account for the width of the marker's corresponding building or location, we keep a buffer in the field of view for the marker. This buffer depends on the distance of the user from the marker. To make sure that the phone is mostly upright, pitch is also constrained to an angle of 45 to 90 degrees.

```
Range<Float> azimuthRange, pitchRange;
azimuthRange = new Range<>(bearing - buffer, bearing + buffer);
pitchRange = new Range<>(-90.0f, -45.0f);

if (azimuthRange.contains(azimuth) && pitchRange.contains(pitch)) {
    markerInRange = true;
}
```

3.2.4. OpenGL

Android includes support for high performance 2D and 3D graphics with the Open Graphics Library (OpenGL), specifically, the OpenGL ES 2.0 API. OpenGL is a cross-platform graphics API that specifies a standard software interface for 3D graphics processing hardware.

(a) **GLSurfaceView**
This class is a View where you can draw and manipulate objects using OpenGL API calls. You can use this class by creating an instance of GLSurfaceView and adding your Renderer to it.

```
mSurfaceView = (GLSurfaceView) findViewById(R.id.surfaceview);
mSurfaceView.setPreserveEGLContextOnPause(true);
mSurfaceView.setEGLContextClientVersion(2);
mSurfaceView.setEGLConfigChooser(8, 8, 8, 8, 16, 0);
mSurfaceView.setRenderer(this);
mSurfaceView.setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
```

(b) **GLSurfaceView.Renderer**
This interface defines the methods required for drawing graphics in a GLSurfaceView. An implementation of this interface is provided as a separate class and attached to GLSurfaceView instance using GLSurfaceView.setRenderer(). The GLSurfaceView.Renderer interface requires that you implement the following methods:

(i) onSurfaceCreated(): The system calls this method once, when creating the GLSurfaceView. We use this method to perform actions that need to happen only once, such as creating the texture and passing it to ARCore session to be filled during update() and preparing the rendering objects.

```
mBackgroundRenderer.createOnGlThread(/*context=*/this);
```

```
mSession.setCameraTextureName(mBackgroundRenderer.getTextureId());

mVirtualObject.createOnGlThread(/*context=*/this,
        "object.obj", "object_texture.png");
mVirtualObject.setMaterialProperties(0.0f, 3.5f, 1.0f, 6.0f);
```

(ii) onDrawFrame(): The system calls this method on each redraw of the GLSurfaceView. We see this method as the primary execution point for drawing (and re-drawing) graphic objects.

```
public void onDrawFrame(GL10 gl) {
    // code
    mVirtualObject.updateModelMatrix(mAnchorMatrix, scaleFactor);
    mVirtualObject.draw(viewmtx, projmtx, lightIntensity);
}
```

(iii) onSurfaceChanged(): The system calls this method when the GLSurfaceView geometry changes, including changes in size of the GLSurfaceView or orientation of the device screen. For example, the system calls this method when the device changes from portrait to landscape orientation.

```
public void onSurfaceChanged(GL10 gl, int width, int height) {
    GLES20.glViewport(0, 0, width, height);
    mSession.setDisplayGeometry(width, height);
}
```

3.2.5. Back-end linking

The app retrieves the virtual markers from the server based on the user's current location. To receive these markers a call is made to the POI helper server where the virtual markers are saved in JavaScript object notation(JSON) format.

(a) **Retrofit**
Retrofit is a type-safe HTTP client for Android and Java. Retrofit turns the HTTP API into a Java interface. It is used to make API calls to the POI helper server. HTTP's response body contains any information and Retrofit parses the data and maps it into a defined Java class.

(i) GSON
The converter dependency provides a GsonConverterFactory class. Retrofit automatically takes care of mapping the JSON data to Java objects (MarkerInfo).

```
mRetrofit = new Retrofit.Builder()
                .baseUrl(mBaseUrl)
                .addConverterFactory(GsonConverterFactory.create())
                .build();

        mMarkerApi = mRetrofit.create(MarkerApi.class);
```

(ii) Request
The URL contains Start latitude, end latitude, start longitude, end longitude to define the geographic region that the markers will be confined to.

```
public interface MarkerApi {
    @GET("36.97398389105355/37.00942677981021/
    -122.08119844562987/-122.0473811543701/")
    Call<List<MarkerInfo>> getMarkers();
}
```

(iii) Call

```
Call<List<MarkerInfo>> call = mMarkerApi.getMarkers();

call.enqueue(new Callback<List<MarkerInfo>>() {

@Override
public void onResponse(Call<List<MarkerInfo>> call,
        Response<List<MarkerInfo>> response) {
            mMarkerList.addAll(response.body());
    }

// on failure code
});
```

(b) **POJO**
POJO means Plain Old Java Object. It refers to a Java object (instance of definition) that isn't bogged down by framework extensions.

```
public class MarkerInfo {

    @SerializedName("_id")
    @Expose
    private String id;
    @SerializedName("time")
    @Expose
    private LocationTime time;
    @SerializedName("name")
    @Expose
    private String name;
    @SerializedName("category")
    @Expose
    private String category;
    @SerializedName("location")
    @Expose
    private MarkerLocation markerLocation;

    // setter and getters
}
```

3.3. Object Display

There are two ways in which the object can be augmented. It can either be stationary and blend with the real world such that you can view it from different angles or it can move along your direction to help with navigation. In this sample app, we use Wood Signboard as our object to augment. In the OpenGL onDrawFrame call, we check if a marker is in the range of our camera direction.

```
if (marker.getInRange()) {
    if (marker.getZeroMatrix() == null) {
        marker.setZeroMatrix(getCalibrationMatrix());
    }
}

if (marker.getZeroMatrix() == null) {
    break;
}

mPose = new Pose(translation, rotation);
mPose.toMatrix(mAnchorMatrix, 0);

Matrix.multiplyMM(viewmtx, 0, viewmtx, 0, marker.getZeroMatrix(), 0);
```

```
mVirtualObject.updateModelMatrix(mAnchorMatrix, scaleFactor);
mVirtualObject.draw(viewmtx, projmtx, lightIntensity);
```

If it is visible for the first time, its zero calibration matrix is set according to the current frame so that it is displayed again in the same location. Translation and rotation matrices are set manually such that every sign board is displayed in a similar way in different directions for its corresponding markers. If the object is needed for navigation then the following lines of code can be added before the pose is created. A sample of the working can be viewed here [8]

```
frame.getPose().getTranslation(translation, 0);
translation[1] += -0.8f;
translation[2] += -0.8f;
```

This takes the translation of the current frame and appends a certain distance to the x and y coordinates so the object appears to be moving along with the user.

4. Backend

As stated in the Introduction section, the goal of the back end is to be an adaptable data warehouse that allows a wide range of geo location based applications to be developed on top of it. Hence the design principle was to be as lightweight and transparent as possible. To be invisible from the application perspective and remain free from any structure, falling inline with modern Rapid Application Deployment (RAD) back end as a service platforms.

To achieve the objectives set, we will use the MEAN (MongoDB, ExpressJS, AngularJS & NodeJS) stack in contrast to more traditional web based technology stacks. Instead of using AngularJS and pushing HTML processing onto front end, we will use HandleBars [9] as rendering engine and process the view at the server. In this process we increase efficiency and we have the flexibility to render an HTML view on mobile & desktop apps as a service directly where they are supported. In addition to the above stack, we also use JQuery, Bootstrap and Google Maps API [10] to interact with elements, remain screen size flexible and allow the developer as well as the users to visualize the marker data correspondingly.

Let us visualize the entire back end by looking at the UML diagram in Figure 2.1 below. The structure of this back end is segregated into a distinct model, view and controller portion making it extremely easy to understand and make changes. The view portion is template based, render-able and requires no coding expertise to give the application a unique feel and aesthetic. Similarly the model will simply accept and respond back with text based JSON objects, and no SQL query knowledge is required.

The entire controller for both the web view and the back end API creator is programmed in JavaScript eliminating any format conversions and efficient communication between them. This is an added benefit for the developer as he does not need to be aware of multiple languages to make small changes and also to make their application schema independent as JavaScript is a free form scripting language.

Some of the special features that the back end will offer will include refresh rate - to account for marker data that is time variant and continuously updated, such as bus location, food truck location and so on. It will also keep reference of the opening/closing time of each marker if that field exists and process through the time data and keep track if the marker is in operation at the current local time. Extension handles will also be added wherever possible to make it easy to add more features, for example: Extension handles for a scoring system on the basis of visiting the markers are included everywhere, as well as dynamic local filtering customization options.
Google Maps Service will be used to render the markers and visualize them as it is the most widely used Geo Mapping Service, but this can be similarly swapped with any other Web based Mapping Service.

4.1. Components:

The back end involves following parts:

1. MongoDB (Model)
2. Express Based Server Controller
3. Front End Client Controller
4. Handlebar Layouts (View)

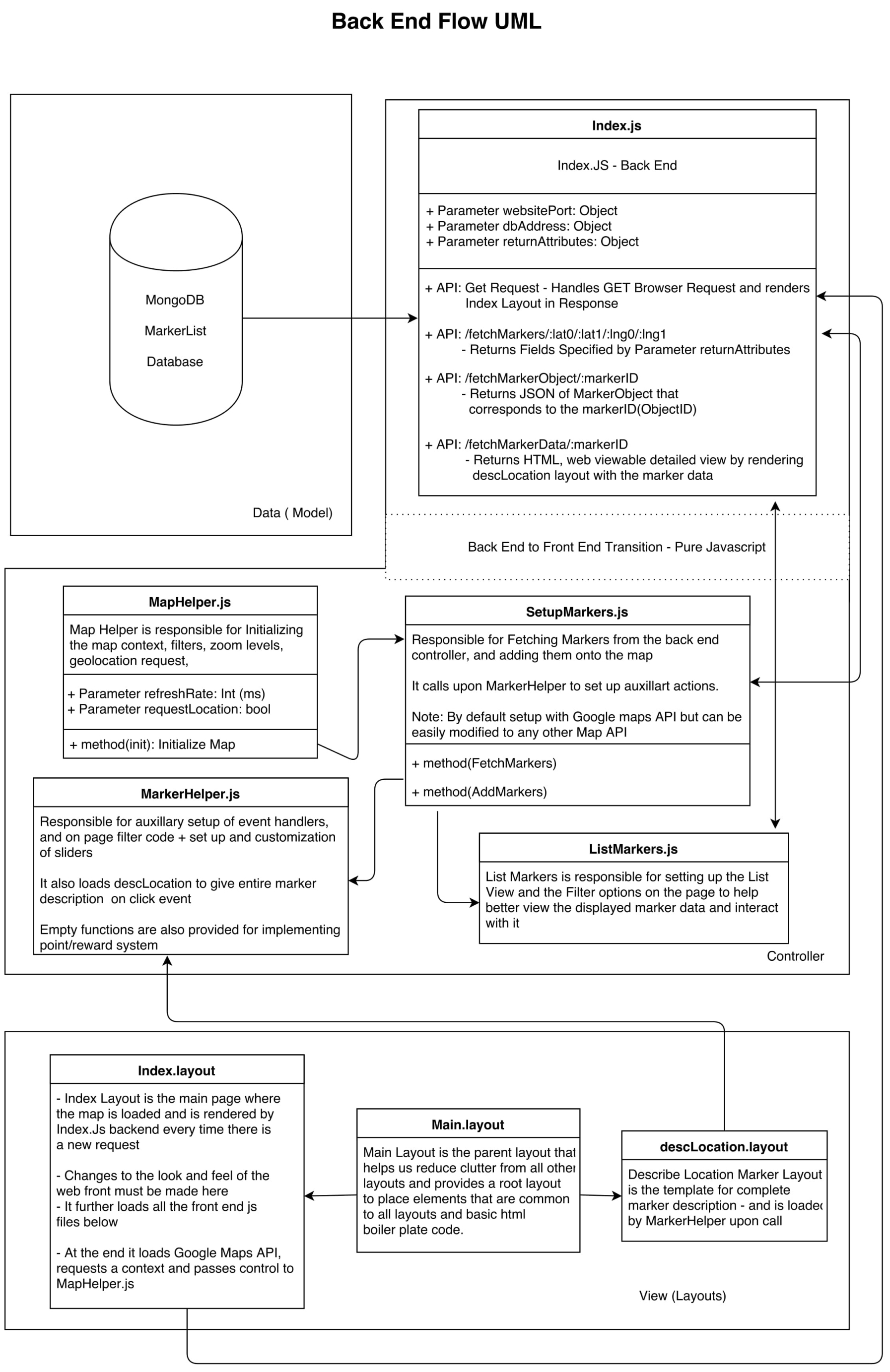

Back End Flow UML
MongoDB
MarkerList
Database
Data (Model)
Index.js
Index.JS - Back End
+ Parameter websitePort: Object
+ Parameter dbAddress: Object
+ Parameter returnAttributes: Object
+ API: Get Request - Handles GET Browser Request and renders Index Layout in Response
+ API: /fetchMarkers/:lat0/:lat1/:lng0/:lng1 - Returns Fields Specified by Parameter returnAttributes
+ API: /fetchMarkerObject/:markerID - Returns JSON of MarkerObject that corresponds to the markerID(ObjectID)
+ API: /fetchMarkerData/:markerID - Returns HTML, web viewable detailed view by rendering descLocation layout with the marker data
Back End to Front End Transition - Pure Javascript
MapHelper.js
Map Helper is responsible for Initializing the map context, filters, zoom levels, geolocation request,
+ Parameter refreshRate: Int (ms)
+ Parameter requestLocation: bool
+ method(init): Initialize Map
SetupMarkers.js
Responsible for Fetching Markers from the back end controller, and adding them onto the map
It calls upon MarkerHelper to set up auxillart actions.
Note: By default setup with Google maps API but can be easily modified to any other Map API
+ method(FetchMarkers)
+ method(AddMarkers)
MarkerHelper.js
Responsible for auxillary setup of event handlers, and on page filter code + set up and customization of sliders
It also loads descLocation to give entire marker description on click event
Empty functions are also provided for implementing point/reward system
ListMarkers.js
List Markers is responsible for setting up the List View and the Filter options on the page to help better view the displayed marker data and interact with it
Controller
Index.layout
- Index Layout is the main page where the map is loaded and is rendered by Index.Js backend every time there is a new request
- Changes to the look and feel of the web front must be made here
- It further loads all the front end js files below
- At the end it loads Google Maps API, requests a context and passes control to MapHelper.js
Main.layout
Main Layout is the parent layout that helps us reduce clutter from all other layouts and provides a root layout to place elements that are common to all layouts and basic html boiler plate code.
descLocation.layout
Describe Location Marker Layout is the template for complete marker description - and is loaded by MarkerHelper upon call
View (Layouts)

Figure 1: Backend Structure UML

]

The interaction between them can be summarized in the UML diagram shown in Figure 1.

4.1.1. MongoDB

MongoDB is a free open source NoSQL/document based database software written in C++. It stores data in JSON-like documents and allows users to query using JSON queries instead of the use of SQL. This allows it to break through the conventions of schema based fixed format RDBMS, increasing flexibility in design with dynamic schemas and making it simple to have complex structures without reducing in performance. It is perfect for marker or location based data by its construction and is optimized for fast read times attributing to in memory search. The drawback or limitation of MongoDB is in terms of complex querying options such as JOIN, trigger etc. In addition the advantage of JSON documents for our application is that is negates the need for serialization & JSON conversion in the API.
Each location marker has the following recommended JSON structure:

(a) ***ID***
12 Byte auto defined unique identifier
  - a 4-byte value representing the seconds since the Unix epoch,
  - a 3-byte machine identifier,
  - a 2-byte process id, and
  - a 3-byte counter, starting with a random value The Object ID is auto generated on insertion and must not be specified in the input object.

(b) **Name** A ”String” specifying the name of the marker at the specified location

(c) **Location** Marker Location Object contains the following attributes:
  - Lat: A ”Float” that holds the latitude information of the marker
  - Lng: A ”Float” that holds the longitude information of the marker

(d) **Category**
A ”String” specifying the type of object. The type string is also used as a filter for category, and to fetch the corresponding marker icon stored in `views/images/markerIcons`.

Special Case: If the Category is Defined as "Sequence", it uses Order and Priority to display a corresponding marker icon.

(i) Order
An "Int" that represents the numbering of the marker for category sequence

(ii) Priority
An "Int" that represents the importance by color coding of marker for category sequence

Note that for efficiency and operation of Category search we have implemented an Index on Category Field in MongoDB.

(e) **Images**
An "Array" that holds either:

- "String" - Simply the Image Address
- ImageObject — Optional
  - Heading: A "String" that is the name of the Image
  - Description: A "String" that stores a 2 line description
  - Likes: An "Int" that stores the number of unique ip likes for that image

(f) ObjMesh — Optional
An "Object" that holds either:

- MeshObj: A "String" that stores the Obj File location
- MeshMTL: A "String" that stores the Mesh Material location
- MeshTexture: A "String" that stores the path to texture image

(g) Ranking — Optional
A "Float" specifying the ranking or order of display of the marker in List View

(h) SubName — Optional
A "String" specifying the line of information that goes along with the Name to provide quick information about the place.

(i) Time — Optional
A "String" that stores the opening and closing hours of the marker. The Time String must be in following format or will be ignored:

```
[Day], ... ,[Day]: [Start Hour].[Minutes]-[End Hour].[Minutes]; \\....\\
[Day], ... ,[Day]: [Start Hour].[Minutes]-[End Hour].[Minutes];

Day Options: Mo (Monday) ,Tu (Tuesday),We (Wednesday), Th (Thursday),
Fr (Friday), Sa (Saturday), Su (Sunday)

Time: Starting Hour & Ending Hour in 24Hour Format
```

Example:
Mo,Tu,We,Th,Fr: 08.00-20.00; Sa: 10.00-18.00; Su: 10.00-15.00;

Notes:

- The Time Field is capable of handing closing times that extend into the next day. For Example if you have a marker that is open from 10AM to 02AM (Next Day) the internal code will compensate for that. You need to simply structure it as:
  Example: Mo: 08.00-02.00;
- The Time Parser in Index.JS parses this time string into a Time Object with following Attributes
  - isOpen: A "Bool" representing if it is Open now in local viewing timezone
  - Data: Formatted HTML string that can be used to display open hours of the marker

  A closed icon must be provided along with normal icon, if not the marker will be ignored when closed.

(j) DataBlocks
An Array of DataBlock Objects, with attributes:

- Heading:
- Data:
- .....

.... Represents the additional marker attributes that are displayed in Marker Description view, they are completely customizable. Can hold both Strings as well as render HTML in the data attribute.

(k) **Visible**
A "Bool" that keeps track if the marker needs to be visible to the user.

Default value is true. SubLocation attribute overrides the Visible property.

(l) SubLocations
An "Array" of ID's of location markers to be displayed when this marker has been clicked. This attribute is only valid if SubLocationFocusMode is true in configuration file.

..... More Objects Can Be Added that can then be fetched with the Fetch Marker Object API call or by Fetch Marker calls if the attribute is also included in ReturnAttributes parameter.

In the above specifications, the attributes written in Bold are Required, if not present the marker will be ignored. In case of formatting errors, the marker will be ignored too. Rest of attributes are free flow and may or may not be included on users discretion, any number of attributes can also be added to the above structure without any change to the back end code.

Example JSON for Object Marker Insertion with Required Attributes:

```
{
    "Name" : "Owl's Nest Cafe",
    "Location" : {
        "lat" : "36.9988554",
        "lng" : "-122.0661359"
    },
    "Category" : "cafe",
    "Images" : [
        "http://eat.ucsc.edu/images/004ce48e.owls-nest.jpg",
        "https://www.placelookup.net/photos/314500.jpg",
    ],
    "Images" : {
        "MeshObj" : "objectFile.obj",
        "MeshMTL" : "objectMaterial.MTL",
        "MeshTex" : "objectTexture.jpg"
    },
    "Time" : "Mo,Tu,We,Th: 07.30-20.30; Fr: 07.30-17.00",
    "DataBlocks" : [
        {
            "Heading" : "Phone Number",
            "Data" : "9819915090"
        },
        ....
    ],
    "Visible" : true
}
```

4.1.2. NodeJS + ExpressJS Server & Back End Controller

NodeJS is an open source asynchronous networking framework that is based on JavaScript and allows us to write dynamic high performance scalable web applications. ExpressJS [11] is a node package [12] that is most commonly used to write API's on top of Node. It is a thin layer add-on that allows for quick and effortless API setup. It also facilitates the use of third party middleware services and middleware rendering engines such as HandleBars.
HandleBars is a micro templating engine that allows us to write quick template HTML files. This allows us to segregate between the data model and the view, allowing quick readable changes.

Because we decided to go with MongoDB, it is the obvious choice to stick with NodeJS and ensure that we do not have interoperability issues by making the entire stack based on JavaScript. Another advantage is that the data structures and operating procedure are the same on both server and client side of the controller thereby reducing confusion. Moving code from back end to front end and vice versa is simply plug and play. It can be moved to achieve performance benefits and to enhance ability to customize as and when deemed necessary.

Let's have a short walk through to setting up the backend:

1 Install NodeJS & MongoDB
2 Setup a new project with npm init, and add required packages namely "express" and "express-handlebars"
3 Git the entire backend such that following file system is available to you:
Root Folder:
    - Index.JS - The complete back end controller, responsible for handling requests, fetching data from MongoDB and responding to API requests.

- Views
  - Layouts - Handlebar Template Layouts
    - Main Layout
    - Index Layout
    - Location Description Layout
  - Css - The CSS files references in the layouts, included by default are:
    - Main.css - Skeleton boiler plate CSS for setting up the page and the basic parameters constant across all pages
    - Index.css - Styling elements only for Index Layout
    - descLoc.css - Styling elements only for DescLoc Layout
    - slider.css - Slider styling for use across the back end
  - Js - The JS files references in the layouts, refer to the flow UML diagram for further information.
  - Images
    - Icons: Icons referenced in the Layouts
    - Marker Icons - Marker icons used to display markers onto the Map
      - Custom Icons For Each Self Defined Category- icon.[format] + closedicon.[format]
      - SequenceIcons: Included in System for Sequence Category
  - DummyData.JSON: Dummy Marker Data with 1 Marker Attribute, this file can be added upon to populate the dataset with more markers
  - Favicons - Favicon Icon for difference screen sizes as referenced In layouts

4 Initiate MongoD service by simply running the program, and load the included JSON data set that generates a Collection with all the Marker Data.

5 At this point, you have all the code setup, you can simply run node Index.JS to launch the service and initialize the API.

To verify the setup you can visit your hosted server address to look at the mobile view with correct marker data that was entered in Dummy Dataset. To understand whats going on behind the scenes better let us look into each component of the UML shown in Table 1.

Index.JS (Server Side Controller)

Index.js loads the needed extensions express and express handlebars, and sets express handlebars as the defined view engine. It also sets the path for views and layout directory.
The following Parameters are used to customize the Server Side Controller:

1. dbAddress - Address where MongoDB is hosted
2. websitePort - Port on which API will launch onto, by default it is 3000. By default browser looks at Port 80. Port 3000 must be specified when loading the website.
3. returnAttributes - Attributes that must be returned for setting up markers. This is done to increase effeciency and so that all attributes are not needed to be returned at setup. By default there are the required markers: var returnAttributes = "Name": 1.0, "Location": 1.0, "Images": 1.0, "Time": 1.0, "Category": 1.0 ; 1.0, indicates to Include them, 0.0 or no mention means to not include them.

All the API's are then setup to respond to GET calls by default, the entire specifications for API calls offered by the controller can be viewed in the table below.
There are two response types in our API:

- Render(LayoutName, JSONObj): Renders the specified Layout with the use of Handlebars. The arguments are LayoutName - String that specifies name of layout file and JSON object that is passed to the layout for dynamically updating the layout

- JSON: Simple JSON object response

The API can be switched to POST calls or any other response method by changing the function parameters if required.

/fetchMarkers/:lat0/:lat1 /:lng0/:lng1/:categories	Required Arguments: • Lat0 — Min Latitude of the viewable Bounding Box • Lat1 — Max Latitude of the viewable Bounding Box • Lng0 — Min Longitude of the viewable Bounding Box • Lng1 — Max Longitude of the viewable Bounding Box • Categories — Which Category Markers are to be returned, multiple marker categories must be seperated by a comma. For Example: Cafe, Dining, Library, ... Returns an Array of MarkerObjects with attributes as specified in the returnAttributes variable of Index.JS
/fetchMarkerData/:markerID	Required Arguments: • markerID — 12 Byte Marker Object ID Returns formatted HTML Description layout - rendered from the Marker Detail handlebar template
/fetchMarkerObject/:markerID	Required Arguments: • markerID — 12 Byte Marker Object ID Returns a JSON MarkerObjects with all available attributes

Table 1

Index.js is also responsible for parsing through the time attribute and generating the time object with isOpen field as described in the specifications Part 2.3.1.h. It works in reference to NodeJS local time and can be updated my changing NodeJS configuration file. At the end of the file we setup the app.listen(websitePort); to initiate the API on the predefined port.

4.1.3. SetupMarker, MarkerHelper, MapHelper & ListMarker (Client Side Controllers)

The client side controllers manage most of the interaction for the back end view and are responsible for loading and updating the model as the user moves the map view frame. As the page is rendered the first controller method to be called is InitMap(), which is a part of MapHelper. Let us look at the functions contained by each controller file and how they interact with each other better.

1. MapHelper: MapHelper holds all the functions that load and interact with the rendered Map.
    - Map, InfoWindow, RefreshMarkers (Parameters): Map Variable holds the Map object, InfoWindow holds an information window that can be overlayed over the map for variety of purposes and RefreshMarkers is a variable that acts as the timer for variety of map purposes such as debouncing user interactions and handling refresh rate functionality
    - RefreshRate (Parameter): Refresh Rate allows user to refresh the map every [xxxx] millseconds. This is particularly useful when dealing with dynamic objects that can change their location with time. Examples of dynamic objects are buses, food trucks etc
    - RequestLocation (Parameter): The Request Location parameter is used to enable user location tracking in web view, if enabled the we make a request for user location and update it every 3 seconds on the Map. User can override RequestLocation by denying the location request
    - InitMap (Method): Init Map function is the first function that is called at the client side controller. It is called as soon as the map is loaded and it gives reference to Map InfoWindow variables as well as sets up interating event handlers. After completing Map rendering it makes a call to SetupMarkers.

- GeoLocate (Method): GeoLocate method is used by InitMap to request permission from the HTML5 Navigator object and request user location and appropriately render it to the Map. Note: GeoLocate will work only if server is hosted in secured HTTPS mode.

2. SetupMarkers: Setup markers is responsible for fetching all of the marker data, modifying its structure as needed and converting them to geomarkers that can be rendered onto the Map. In addition to that it also calls upon ListLocation methods to add the marker to the list and maintain a dictionary of objects.

    - LoadMarkers (Method): The Load Markers function makes a GET call to the API to fetch all the markers that exist in the specified latitude and longitude bounded box. It does not return the complete marker information but only the fields that are specified in the returnAttributes Parmaeter. In case of user interaction/map movement, the load markers compares the newly loaded markers with old markers - retains repeated markers while deleting markers that no longer exist in view efficiently without a complete redraw. It also maintains a dictionary with the latest marker data and their locally assigned id's.
    - LoadCategories (Method): The Load Categories function makes a GET call to the API to fetch all the categories that exist within the requested geolocation bounded box specified by latitude and longitude. This is useful to display a category filter bar.
    - AddMarkers (Method): The Add Marker function is responsible for converting JSON marker data to a GeoMarker that can be added to the Map. This function is called iteratively by LoadMarkers to add each marker to the Map.
    - ClearMarkers (Method): The Clear Markers method is responsible for destroying the GeoMarkers from the Map that are no longer visible, to save rendering resources.

3. ListLocations: List Location manages the List View of all the Markers visible in Desktop Mode. It is also capable of setting up a Dat.Gui based filter system.

    - FilterObject (Parameter): The Filter object allows the developer to setup local filters on various attributes dynamically. This is specifically helpful for reordering markers, or narrowing the attribute range for example: price. It can be used to filter any marker contain the specified numerical attribute.

   - AddObjectToList (Method): AddObjectToList is called upon by the AddMarker function to add the marker information to the list view at the same time when rendering it to the Map. It adds the object to the list in HTML format specified in the file itself using string manipulation and DOM conversion. This is done to improve efficiency and to avoid making repeated server calls.
   - SetupFilter (Method): SetupFilter function parses the Filter Object and appropriate renders the filter controls on the screen, and sets up the interaction event handlers that can affect the marker dictionary and adjust marker objects.

4. MarkerDetails: The MarkerDetails controller loads and handles the interactions responsible for displaying the complete Marker data. In desktop as well as mobile view, the MarkerDetails is launched when the marker is clicked on the Map. In addition to that in Desktop view the MarkerDetail window can be opened by clicking on the List View marker block.
   - OpenMarkerDetails (Method): The OpenMarkerDetails is called upon marker or list view marker block click event and makes a GET call to the fetchMarkerData API. The FetchMarkerData API returns HTML rendered DetailWindow with ObjectID specified. It makes use of the MarkerDetails layout to render the data and overlays it onto our web page.
   - CloseMarkerDetails (Method): the Close Marker Details is called upon a stray click or Back button to close the overlayed Marker Detail Window.

4.1.4. Layouts (View)

The layouts are located in the layouts directory and are essentially handlebars based render-able layouts that are used by the back end controller to deliver HTML content back to service user's requests. There are essentially three layout files and they have the following structure:

1. **Main Layout**: The Main Layout contains boiler plate HTML content that is required for rendering any HTML content. It is loaded upon any new request and has the following structure:

```
<!DOCTYPE html>
<html style="width: 100%; height: 100%;">
    <head>
        <meta charset="utf-8">
        <link rel="apple-touch-icon" sizes="57x57" href="favicons.ico/
        apple-icon-57x57.png">
        ... Favicons For Different Screen Sizes ...
        <meta name="theme-color" content="#ffffff">

        <link href="https://fonts.googleapis.com/css?family=Lato:100"
        rel="stylesheet" type="text/css">
        <link rel="stylesheet" href="css/index.css" type="text/css">
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/
        Swiper/4.0.3/css/swiper.css">
        <link rel="stylesheet" href="css/slider.css">

        <script src="http://ajax.googleapis.com/ajax/libs/jquery/
        1.11.3/jquery.min.js"></script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/Swiper/
        4.0.3/js/swiper.min.js"></script>
        <script src="js/slider.js"></script>
        <title>UCSC GEO AR</title>
    </head>
    <body style="width: 100%; height: 100%;">
    {{{body}}}
    </body>
</html>
```

2. **Index Layout:**
   The Index Layout is the default webpage that the loads up on the users browser as they visit the server address or as they load the domain. The Index Layout is responsible for rendering the Map View and contains all of the above specified front end controllers. We use a one web page model, hence everything is loaded onto a single page. There is no user redirection by default.
   The layout has the following design structure:
   - TopBar: The Topbar by default holds only the Logo from Images/logo.png and the rest of the blank space is for user to customize and in future may hold further filtering options
   - Category Bar: The category bar dynamically loads all the categories in the current view frame, and allows users to filter through them
   - Map View: The Map View renders the Map Service and loads Markers onto it
   - List View: The List View holds MarkerBlocks, that allow user to look at 5 top Images of the Marker, and to select the marker from it

   The Map view above covers 100% of your screen width leaving aside the List View of size 400pixels for a desktop browser, whereas the top bar and category bar have the same height of 60 pixels (Figure 2).
   On Mobile view (Figure 3), the same web page fluidly restructures to fit just the map view initially. The user can toggle between the map view and the list view within the 400px view for the small width mobile browsers.
3. **Marker Details Layout:**
   The marker details layout is displayed below and consists of following components:
   - Marker Image Sliders: A slideshow of all the images specified for the Markers
   - Marker Name & SubName: Marker Details
   - Direction in Maps: It utilizes the Google Map API to load the directions between the users current location and marker. If user access's the website on mobile view it opens the google maps app if it is already installed to further simplify operations

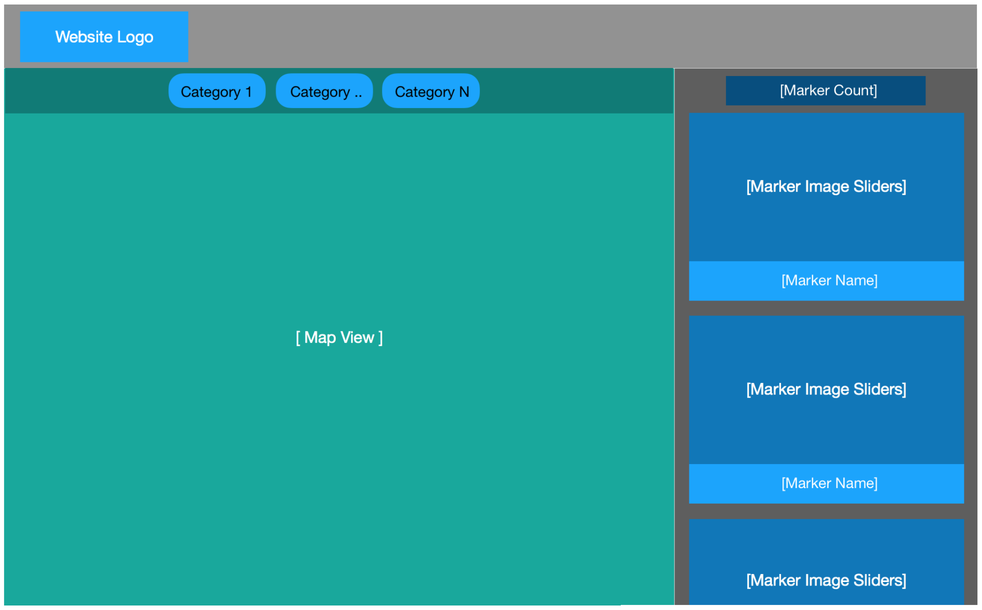


Figure 2: Desktop View

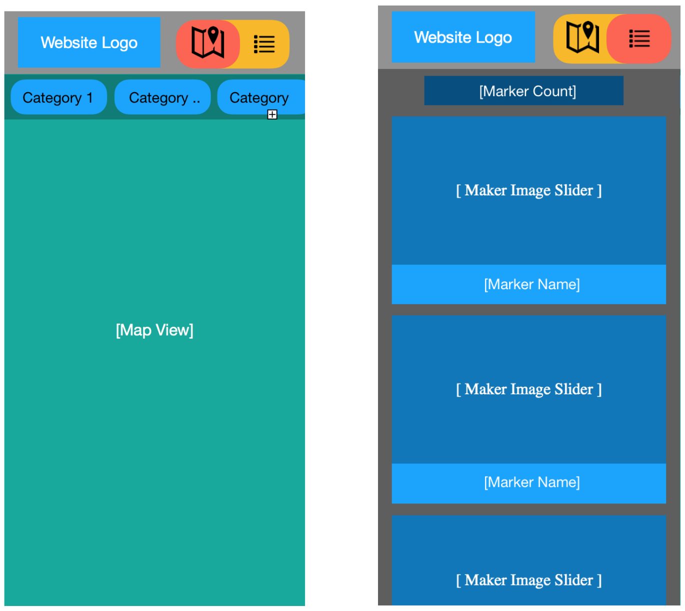


Figure 3: Mobile View, the first image corresponds to the Map view version that renders all the markers on the map whereas the list view version is shown on the next image. The switch on the top allows users to toggle between Map View and List View.

- Blocks: Blocks allow various type of textual content to be displayed, the Data in blocks can be either text data or HTML data that is rendered onto the page

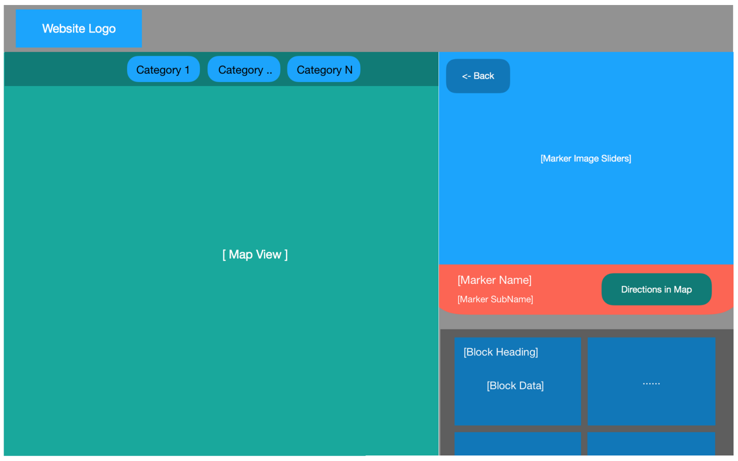


Figure 4: Desktop Marker Detail View

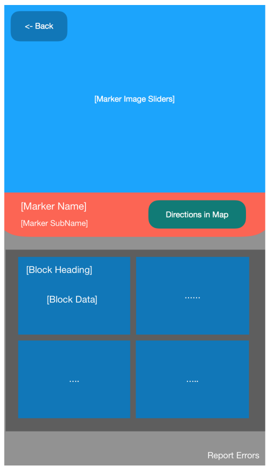


Figure 5: Mobile Marker Detail View

5. Results

The results we obtain can be best understood by seeing the live demo screen shots that are added below for each three test platforms.

5.1. Android Demo Application

The app fulfills all of the expectations and goals that were set in the initial stages of development. Since the app depends on GPS and sensors, its behavior is variable and not always consistent with the expected results. As Google ARcore improves its understanding of the world, the rendered object's pose will be more stable. A short video of the working can be viewed on YouTube [13] and the code is available on GitHub [14].

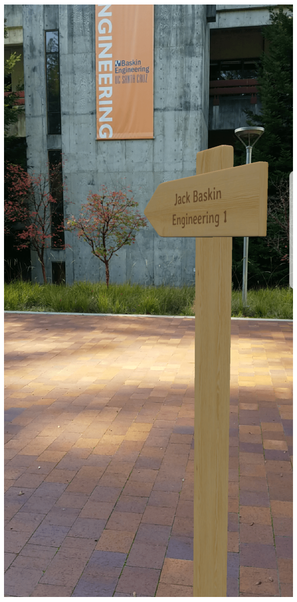


Figure 6: Jack Baskin Engineering 1

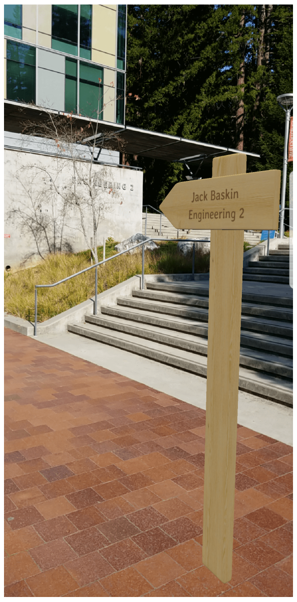


Figure 7: Jack Baskin Engineering 2

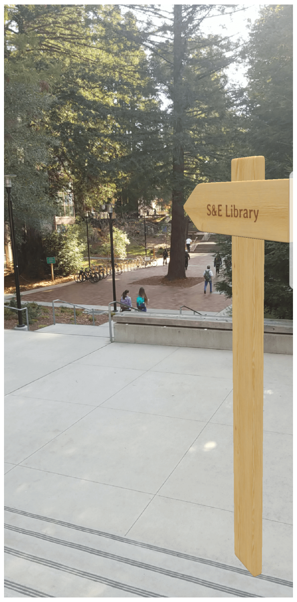


Figure 8: Science and Engineering Library

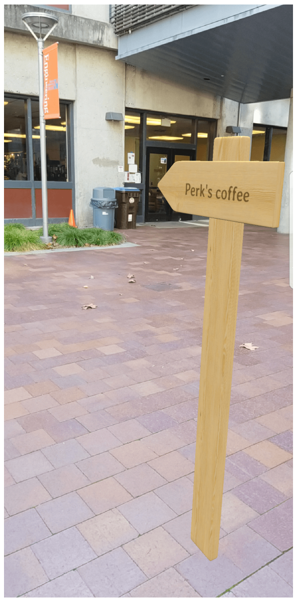


Figure 9: Perk's Coffee

5.2. Web Based Application

The web app makes a great place to start experimenting with the back end structure. It can also be used to verify the API operation with the use of live formatted console logs.

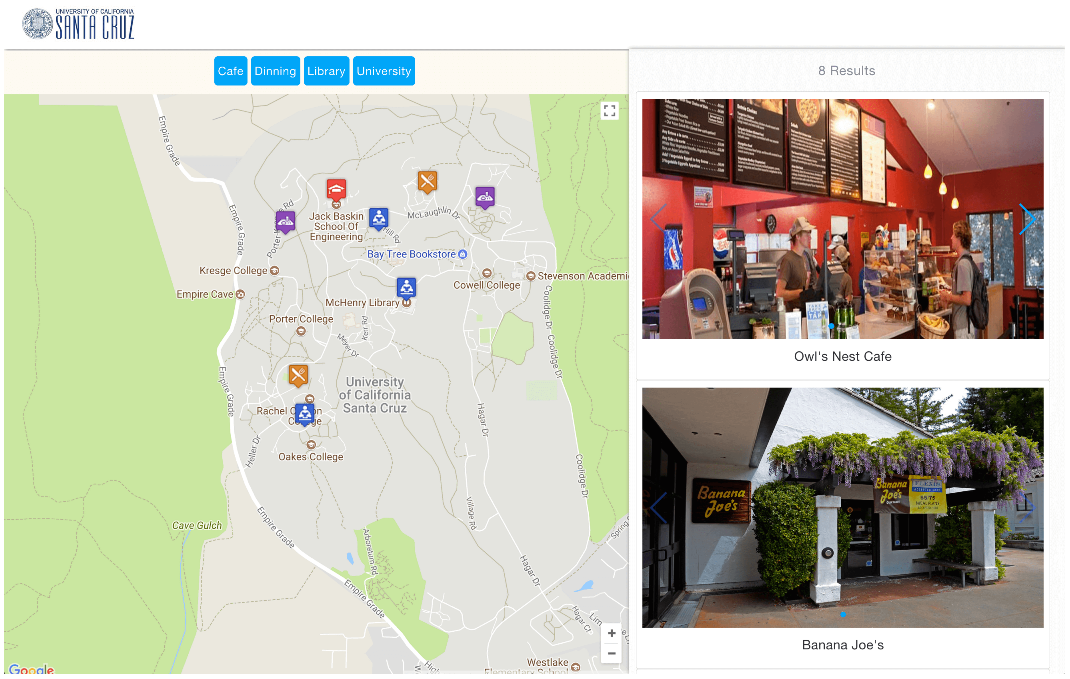


Figure 10: Desktop View

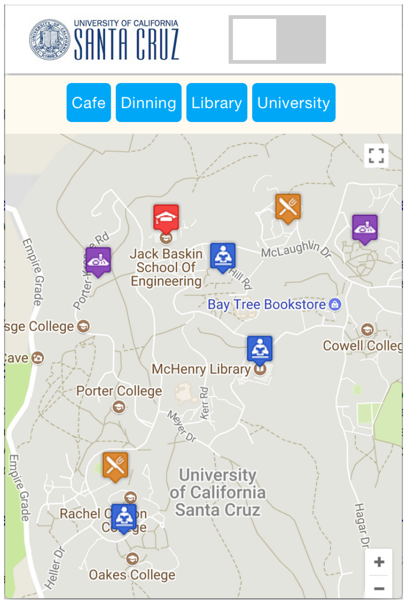


Figure 11: Map View

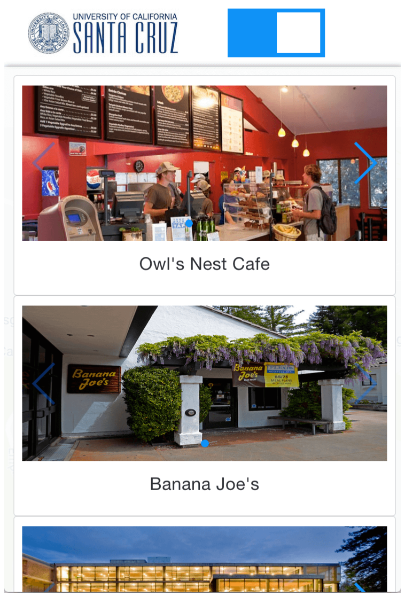
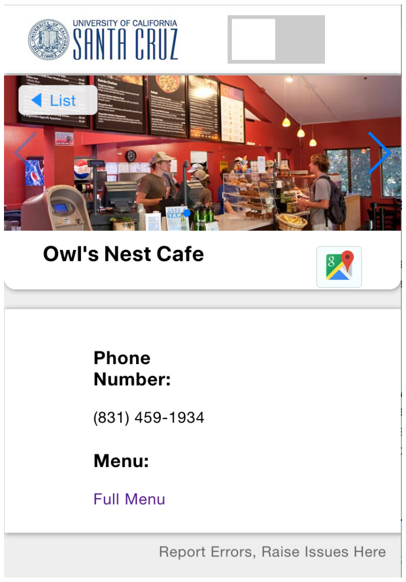


Figure 12: List View



Owl's Nest Cafe

Phone Number:

(831) 459-1934

Menu:

Full Menu

Report Errors, Raise Issues Here

Figure 13: Detail View

5.3. iOS Annotation Demo Application

In this example, the backend passes geo-location markers to iOS Application. The iOS Application created as a demo, simply parses the marker and converts it into a label and then forwards it to the Annotation View Framework for rendering. Instead of a label a 3D object can also be displayed in a similar fashion.

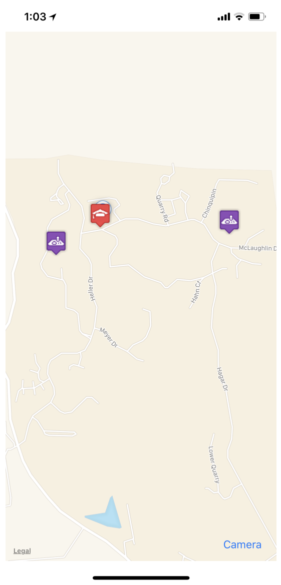


Figure 14: iOS Map View

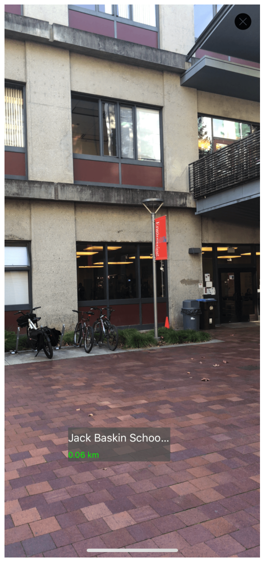


Figure 15: iOS AR View Pointed Towards Jack Baskin College

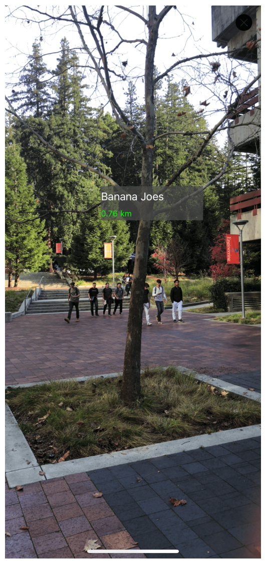


Figure 16: iOS AR View Pointed Towards Banana Joes

6. Conclusion and future work

The app is able to fetch virtual markers from the back-end server, render 3D objects in the user's camera view based on marker locations and display additional information about the point of interest. The need to make it generic and able to fit to a variety of applications required thorough brainstorming. Also working with the brand new AR libraries namely ARKit on iOS and ARCore on Android to understand their internal workings and ensure that one service is compatible with both, drove the need to try multiple approaches and settling on the best. A considerable amount of literature review and a good deal of learning about servers, authentication, API creation and MongoDB based NoSQL data stores has been put into this project and we look forward to see other developers using this platform to create unique and fun AR apps.

The app and the corresponding back-end serve as a good foundation to create any Geo-location based AR app. Since the app was built using a preview version of Google ARCore, once the version 1.0 is released, the project would need to be updated.

References

[1] "Latitude and longitude finder," https://www.latlong.net/.

[2] "Google arcore: Getting started with android studio," https://developers.google.com/ar/develop/java/getting-started.

[3] R. P. J. S. M. R. Philip Geiger, Marc Schickler, "Location-based mobile augmented reality applications challenges, examples, lessons learned," *Institute of Databases and Information Systems, University of Ulm, James-Franck-Ring, Ulm, Germany.*

[4] "Yelp monocle: Support page," https://www.yelp-support.com/article/What-is-Yelp-s-Monocle-feature.

[5] "Google android api guides," https://developer.android.com/guide/index.html.

[6] "Google arcore api reference," https://developers.google.com/ar/develop/java/getting-started.

[7] "Outware insights which direction am i facing?" http://www.outware.com.au/insights/which-direction-am-i-facing\
-using-the-sensors-on-your-android-phone-to-record-where-you-are-facing/.

[8] "YouTube android geolocation augmented reality using google arcore (2)," https://youtu.be/5TBL0sHFRzw.

[9] "Express handlebar source and documentation," https://expressjs.com/.

[10] "Google maps api documentation," https://developers.google.com/maps/.

[11] "Express server source and documentation," https://developers.google.com/ar/develop/java/getting-started.

[12] "Node package manager tutorial," https://www.npmjs.com/.

[13] "YouTube android geolocation augmented reality using google arcore," https://youtu.be/RAg6u2AZ1fI.

[14] "GitHub geolocation-arcore repository," https://github.com/aishnacodes/Geolocation-ARCore.