– Independent Work Report, Spring 2018 –

Mobile AR Software for Cultural Heritage Documentation

Zachary Liu
Advisors: Professor Branko Glisic and Rebecca Napolitano

Abstract

The proper documentation of cultural heritage projects is important to maintaining the integrity of building infrastructure. Currently-available commercial tools require making tradeoffs between speed, ease of use, and level of detail in their ability to capture and share information. With the recent rise of augmented reality tracking technology on mobile devices, we explore how AR can be used for cultural heritage documentation. We describe the design and implementation of a prototype which presents several tools for capturing and visualizing annotations in world space. We also develop and evaluate algorithms for combining the local precision of AR tracking with the global accuracy of GPS. We conclude that while the quality of current AR technology is still lacking, there is significant promise for future applications.

1. Introduction

The objective of this project is to develop a system for creating, accessing, and visualizing cultural heritage documentation using augmented reality on a mobile device. Existing infrastructure in the U.S. is deteriorating, and to ensure the ongoing safety and integrity of structures, on-site inspections and monitoring are increasingly in demand. These documentation projects produce large amounts of of raw and analyzed data, but current software tools are subpar and difficult to use. Commercially available software for panorama-based virtual tours are inflexible in their ability to visualize the scene and cannot be used on a mobile device, while photogrammetric and other 3d approaches are too time consuming and expensive for many cultural heritage projects. To address this problem, this project seeks to explore the possibility of developing mobile software for performing documentation in augmented reality. We aim to develop a system which can help researchers collect and visualize data using a mobile device, enabling intuitive on-site data entry and updates.

It is important to note that the scope of this overall project is, intentionally, designed to be beyond the scale of one semester. A future portion of the project, to be done by another student, will build upon this code developed during this portion in order to explore the idea further and refine the application. With this in mind, the specific goal of this paper is to develop and evaluate a framework and prototype in order to investigate the practicality of an augmented reality approach. We pay special attention to implementation details and pitfalls, and in explaining the decisions we made in designing for augmented reality, in order to inform future research on this topic. We also present a quantitative analysis of results from our world-alignment algorithm.

2. Background and Prior Work

When recording documentation of cultural heritage, a researcher must define the scope of information it wishes to collect. The scope of a project depends on its goals as well as budgetary and time constraints. Depending on this scope, the researcher can focus on acquiring different levels of precision and accuracy in their data collection process. Letellier divides the scope of cultural heritage documentation projects into three levels of detail: *reconnaissance recording*, *preliminary recording*, and *detailed recording*. [15]

At the reconnaissance recording level, projects in this range aim to collect just enough information to capture the "overall characteristics" of a site, with enough detail to identify and visualize significant areas. Data collection at this level is not performed to scale and often uses a combination of hand-drawn sketches and photos from various points. If a project requires a higher level of detail, preliminary recording techniques are chosen instead. This level of data collection is performed to scale, often using measured drawings and aligned photographs. At the extreme end, detailed recording is performed when a project aims to communicate accurate information for construction, design, or other purposes where this is necessary. This type of recording relies on expensive 3d capture and modeling techniques in order to provide the highest possible level of detail.

We focus our project on tackling the needs of the reconnaissance and preliminary recording stages. These types of recording are useful for planning and initial documentation of structural

conditions, such as recording data on calcite deposits in an archway for later analysis. A project at the preliminary recording level requires collecting data to scale, but only demands a level of accuracy within approximately 2-10 cm, depending on the needs of the specific application.

Current techniques for digital cultural heritage documentation take a number of forms. Hassani divides these techniques into three categories: image-based, non-image-based, and combinative approaches.[7] Within the image-based category, there are two broad methods of capturing image data. The simplest and easiest approach uses panorama photography to capture the cultural heritage site from various perspectives and angles. As Hassani notes, a key benefit of this approach is its compactness, as it is an efficient means of capturing data about a large object with a small number of panoramic photographs. Commercial software, such as Kolor Panotour Pro, is available for constructing "virtual tours" from panoramas.[18] Prior work has shown success in utilizing this software as a system for cultural heritage documentation, as it offers the ability to insert rich annotations on top of captured images. The researcher can use "hotspots" in the virtual tour to link between multiple spherical panoramas around a building, or to link to supplementary data, such as spreadsheets of collected data. The maturity of the off-the-shelf software available in this area is a particular benefit of choosing this approach.

However, a disadvantage of the panoramic approach is the limited freedom of viewpoints it gives to the end user. Each panorama must be viewed from the exact point it is captured, and it often requires standalone viewing software running on a laptop or desktop, limiting its usefulness in the field. Napolitano notes how this limitation makes it inefficient to record information, since the researcher must take the spherical image on-site and bring it back to a computer for editing. [19]

In contrast to the panoramic approach, a more detailed method involves capturing a 3d model of the structure. This could involve using either photogrammetry (an image-based technique) or laser/structured light scanning (a non-image-based technique). Both techniques capture an enormous amount of data, requiring infrastructure to support large data files, but they provide the user with a full 3d model of the structure which can be viewed from various angles and analyzed off-site. It is also very accurate, which makes this technique suitable for measuring changes to a building

over time. In particular, photogrammetry can produce a detailed, textured 3d model which is nearly identical in shape and appearance to the actual object. Unfortunately, this level of detail results in large data file sizes and expensive, time-consuming scanning processes. A prior study comparing a virtual tour panorama approach with a 3d modeling approach found virtual tours to be 16 times faster and 30 times cheaper to produce. [19]

Prior research has begun to explore the feasibility of an augmented reality approach using mobile device cameras. Langlotz et al. developed a novel system which used template-based matching of panoramas to compare the location of a mobile device with a database of existing annotations, and then displays matching annotations at the appropriate locations in the scene. In order to relax the limitation of viewing a panorama from a fixed location, their template-based approach is robust to slight perspective changes, so annotations can still be placed in their proper locations even when the user is not in the exact original location. [12] Lee et al. developed a collaborative framework which combined both mobile-based and desktop-based editing tools into one system. Their framework involves capturing a panorama on a mobile device, moving it to a desktop to place annotations that would be cumbersome to create on a small screen, and then moving it back to the mobile device for in-situ refinement of the annotation placement against the actual building. [14]

These frameworks have in common their use of panoramic photographs to capture the scene. This approach ensures high fidelity data collection, since panoramas are well-tested as as means of accurately capturing documentation. These projects also show the benefits of a client-server architecture. Having a server-side connection provides a means of storing a large database of potential matches. A cross-platform approach which combines both a mobile client and a desktop client also has the potential to realize greater efficiency, since each input method is uniquely suited for certain scenarios. On a mobile device, the user can more intuitively match the information on screen with the location of objects in the real world, while on a desktop, they can more precisely insert annotations.

A key difficulty faced by these projects, however, is the problem of tracking the user in 3d space using only an off-the-shelf mobile device without specialized hardware. Until recently, outside

of the research lab, most AR frameworks for mobile devices focused primarily on marker-based tracking, which involves locating a special tracking image placed in the scene. This is acceptable for certain situations where the user can set up these trackers ahead of time. [22]

In the past year, new augmented reality toolkits for iOS and Android have emerged which have brought high-quality AR tracking to mainstream mobile phones, making new AR applications possible. In June 2017, Apple released the ARKit augmented reality framework, which supported native augmented reality support on recent iOS devices. Soon after, in August 2017, Google released ARCore, which offered similar functionality for Android devices. With both major mobile platforms now offering native AR capabilities, there is high potential in this space. [4]

Both ARKit and ARCore offer a similar set of functionality. Of particular interest to us is their ability to perform pose estimation, which involves predicting both the position and rotation of the device in 3d space. This works by combining data from inertial sensors, including the built-in accelerometer and gyroscope, with visual tracking using the camera. As the user moves their device around the environment, these frameworks track the movement of key points in the scene in order to measure how it is moving. We can view this raw point cloud as a set of 3d points currently visible in the scene. Since this point cloud gives the device knowledge of the surrounding geometry, both frameworks also support the ability to perform *raycasting* into the scene. We can query the framework with the coordinates of a point on the screen and receive back the closest 3d point directly in front of the camera beneath the chosen point.

While these frameworks offer high-precision positional tracking, one limitation is that due to their use of relative positioning from one camera frame to the next, they can be subject to drift over time. Incremental errors build up, especially if the user is moving a long distance. Thus, we can describe AR tracking as being high precision but low accuracy.

Combining AR tracking information with GPS-provided location is one potential solution for this issue. Since GPS sensors provide an absolute location, they are not subject to drift in the same manner as AR tracking. In contrast, they have lower precision. On a mobile device, GPS is typically only accurate to within a few meters at any given time. [28]

3. Approach

The key idea of this project is to develop the annotation platform on top of a cross-platform AR framework, and to integrate this system with a real-time client-server architecture. This enables a unified development approach, where the same codebase can be shared across multiple platforms, including mobile devices and computers. This would help ensure a seamless experience as a researcher transitions from capturing data in the field, to viewing data in the office, and back to adding additional data in the field. By integrating the system with a real-time data server, we are able to not only sync recorded annotations as they are captured, but also able to improve localization accuracy with server-side computations. We can perform computations which would be infeasible directly on the mobile device, whether due to processing constraints or due to data storage demands.

4. Implementation

We used Unity as our cross-platform development environment of choice. It supports building applications for several platforms, including Android, iOS, web, and desktop, all from the same codebase, which fit our requirement. Unity uses C# as its primary scripting language, which was used for our client-side application code. The use of this common language and development environment meant that we were also able to utilize several third-party libraries for certain features. In particular, we used Unity's experimental ARInterface library to support cross-platform AR features. This library provides an abstraction over the ARCore and ARKit frameworks and connects both with a common C# Unity API. We also used the Mapbox SDK to support our mapping functionality.

We implemented the server in Python in order to benefit from the wide range of data analysis tools built for this language. By designing the server in Python, our framework is better suited for rapid development and prototyping. The server and client are connected via the Socket.IO real-time communication framework. Originally designed for web applications written in Node.js, the Socket.IO protocol now works on a wide range of server and client languages and platforms. We originally implemented the server protocol using raw TCP/UDP sockets, but later switched

to use Socket.IO because its built-in support for connection management and data broadcasting. Socket.IO automatically handles a dropped connection by reconnecting to the server, and it also has the built-in ability to broadcast messages from one client to all other clients.

4.1. Initial Experiments

We initially created a prototype which directly accessed the point clouds returned by the AR system in an attempt to reconstruct a full 3d representation of a scene. This approach was inspired by photogrammetry, which normally requires more computationally intensive processing to produce a high quality model. The goal of this initial approach was to investigate whether a lower-fidelity photogrammetric technique might be possible by merging point cloud data from multiple frames. We found that the point cloud data returned by the AR system was very sparse, and the density and quality of the points it found varied dramatically based on the texture of each surface. Indeed, the point cloud returned by the AR system is not intended to be used for further processing. On Apple's ARKit documentation, it warns: "ARKit does not guarantee that the number and arrangement of raw feature points will remain stable between software releases, or even between subsequent frames in the same session ... the point cloud can sometimes prove useful when debugging". [2]

Given that the goal of this project was to create a cross-platform system in which we could view the captured data off-site, it is important that the visual representation we choose to communicate the annotations is accurate. We determined that significant amounts of additional processing would be necessary to transform the raw point cloud data into a 3d model detailed enough for offline viewing. As a result, we decided not to pursue this approach further.

We shifted the project to focus instead on capturing images directly from the camera and saving the pose of these flat images in 3d space. This would better preserve the fidelity of the original annotations, and ensure that the annotations would be viewed accurately.

4.2. Design Principles

In designing how the user would interact with the app in augmented reality, we needed to consider new design principles this alternate medium. In contrast to a standard mobile application, augmented

reality applications operate in an interactive space, where digital overlays must interact with objects in the real world. This presents new UX challenges. Olarnyk describes AR as an "edge case" for UX design and offers three tenants of AR design:

- Real world + digital world
- Has a flexible immersion level
- Interface beyond a screen [20]

These tenants encourage a design approach that thinks carefully about how to manage the interaction between the screen and the physical space. Interacting with objects through the camera can be awkward, and it is important to decide which controls or components should sit on the screen space. Placing user interface elements into the world space should be preferred when this provides greater control or context to the user and makes the user more productive.

4.3. Client Architecture

In creating the architecture for this application, we adapted the common Model-View-Controller (MVC) design pattern for use within Unity. This is a pattern for data-heavy applications which focuses on establishing clear separation of concerns between sections of the application codebase. In the MVC pattern, the "model" corresponds to classes which solely handle data, while the "view" corresponds to classes which display data to the user. In between, the "controller" manages fetching data from the model and displaying it on the view. It also listens to events on the view, such as a button being pressed or a text field updated, and makes any necessary changes to the model. This approach works well for data-heavy web applications, where the primary role of the controller is to manage shuffling data back and forth. But in an interactive 3d environment with complex logic, additional abstractions are needed. [3]

Since Unity is commonly used as a game engine, its development structure favors a visually-oriented, component-based approach. In Unity, an application consists of one or more Scenes, each of which is composed of a nested tree of GameObjects. Every GameObject has a single parent, with those at the top level having the root as its parent. This rigid hierarchical structure maps well to

games or other 3d applications. To add behaviors to this tree, Unity supports an abstraction called components, which are scripts, behaviors, or characteristics which are attached to a GameObject in the tree. They can optionally contain references to GameObjects and/or components anywhere else in the tree, whether further above the hierarchy or further down. Importantly, components are designed to be composable abstractions. For example, we created a drawing component that could be attached to a plane to add drawing functionality to the object.

We combined these types of abstractions in building the application. We treated controllers as a type of component which had more complex logic and which "owned" a piece of the GameObject hierarchy. We then stored the majority of our critical logic in controllers and had them manage the job of loading and saving the model. These controllers are tied to a certain view hierarchy, so we used Unity's "prefab" feature, which provides a way to bundle together and instantiate a subtree of GameObjects and its associated components.

Since a controller owns its subtree, a controller may include or create other controllers as children it manages. Similarly, a component may create and own subcomponents if needed. But with the number of controllers and components which interact simultaneously in the framework, we found it necessary to establish a guideline for how one controller or component may reference and interact with another in order to reduce complexity. When one script decides to call a method on another script, it introduces a hard dependency from the caller to the callee. Consider how this would affect the development of a certain UI element. If we were to structure our code such that this button directly calls a method on its parent controller, then we could never test the button in isolation without the parent. Instead, we use the Observer pattern, named by its use of "observing" event listeners that passively listen for actions on another object. When something should happen in the parent in response to something in its child objects, such as a button triggering an action, we prefer to have the parent controller or component register an event listener on the child. When the button is pressed, the event is triggered by the child by looking for any registered listeners and calling them in turn. The child thus is not dependent on which behaviors, if there are any at all, might be triggered in response to something it does, removing this circular dependency.

Finally, there are certain top-level objects which provide functionality used throughout the application. The most important is the ArController, which is a modified version of the ARController class from the ARInterface library used to provide cross-platform AR functionality. This service initializes AR tracking in the background and streams video from the camera to the background of the scene. We modified the original ARController class to support a secondary object for tracking camera pose, in order to support custom tracking functionality.

4.3.1. Core Controllers The core annotation functionality of the framework is contained within the AnnotationController class and its dependant components. This controller manages all the state for a particular annotation, including whether it is active at the time, which tool is currently activated, and how the annotation should be displayed and rendered. It is backed by an underlying state machine

While the AnnotationController keeps track of the various states it can be in, such as whether it is in Anchor or Draw mode, and which actions are valid in which states, it does not directly manage the UI for these states. Our design uses a single set of UI controls for all annotations, the target of which depends on which is selected at the time. Based on our design principle that no controller should directly make changes to controllers above it in the hierarchy, we implemented this shared UI by wrapping it in a separate controller, ToolboxController.

The ToolboxController acts as a connector between the shared annotation UI and the currently-active annotation. Since there is limited space on a mobile device, we chose to model our app's control system after the kinds of tools found in mobile camera apps. When the user switches into annotation editing mode by tapping on an annotation in the scene, we activate a set of shared tools which includes buttons on the upper right and a close button in the upper left. When the user then chooses a tool, the ToolboxController switches modes to hide the other tool options, and for certain tools displays an action button at the lower right. The underlying state for this modal UI is controlled directly by the AnnotationController via a set of linked callbacks, and the ToolboxController itself manages no state other than knowledge of the current annotation.

MainController manages the top-level state of the application, such as the capture UI and the

logic for creating new annotations. Its state machine currently transitions between two possible states: Camera, in annotation capturing and editing mode, and Map, when the map is open at full size. When the user presses the "capture" button, the controller takes a new photo, saves the current AR-reported pose, and uses this to create a new AnnotationModel object. It then spawns a new AnnotationController instance with this model object and adds this instance to the scene. The MainController then attaches event listeners to track when the annotation enters a "selected" or "deselected" state and to update the UI accordingly.

This MainController also has the responsibility of receiving new annotations from the server and adding them to the scene. At startup, the MainController registers itself as a listener for new annotation objects. This listener is called with any new annotations as they are made.

4.3.2. Models All models in our implementation derive from a shared class, BaseModel, which provides common functionality such as loading and saving data to/from the server. The model object contains both ToJson() and FromJson() methods which, respectively, export data out of the object and update the object with new data. We use the JSON file format when serializing data to/from a string for the purpose of communicating with the server, as this format is human-readable and widely supported on different platforms.

To help simplify the process of syncing data, our framework identifies each model object with a GUID (globally unique identifier). This is a 128-bit identifier, represented by a string, which is randomly generated for each new object. Unlike a typical incremental numbering scheme, the use of a random identifier of this size means that there is practically zero chance of collision with a previously generated GUID, and thus this is a standard practice for generating unique IDs without requiring collision detection. This streamlines our design, since it allows us to generate unique object IDs locally even before communicating with the server.

The AnnotationModel class is used to store all data related to each annotation. It derives from BaseModel and contains the following data fields:

- *CameraPose*: position and rotation of the captured image
- *Image*: ImageModel instance representing the image data

- *Fov*: vertical field of view of the image, in degrees
- *IsAnchored*: whether the image is anchored to a position in the scene
- *AnchorPosition*: where the image is anchored
- *DrawLines*: list of 2d line drawings
- *SurfaceDrawLines*: list of 3d surface line drawings

The ImageModel class mentioned above acts as an abstraction over various image representations. Since an image contains a large amount of data, we do not want to send the entire image to each new client immediately upon load. Instead, we want to wait until the user activates the annotation by selecting it before loading the image data from the server.

4.4. Image Visualization

In order to display a captured image, we needed to decide the best approach for visually representing a captured image and its pose in the scene. The representation would need to convey information about the original position and rotation of the camera, as well as provide a clear indication of the subject of the image. Since the image representation is in 3d space, we also needed to make a decision about the size of the image we display. Each approach is described below, with its benefits and drawbacks.

- *Don't show the image directly in the scene, and instead represent the capture point with a marker of some form.* While this is the easiest approach, this would not give us any of the benefits of representing the image in augmented reality.
- *Place the image at the location of the camera when it was taken, at a constant size.* This was the most straightforward approach, and was the first version tested. However, we found that it was uninitutive for viewing the content of the image. In order to see the 3d image, you needed to walk back from the original capture point
- *Place the image at a location a fixed distance in front of the camera, with proportional size.* Here, "proportional size" refers to a size such that when the resized image in 3d space is viewed from the original capture point, the image aligns with the view from the camera. [diagram] This

sizing approach has the benefit that it can convey information about the original perspective of the camera.

- *Place the image at the distance of the image subject, with proportional size.* This has the same sizing benefit as the previous approach, with the added benefit of more accurately representing the size of the target. For example, when photographing a wall of a building, this approach would scale the image more closely to the size of the original wall.

We ultimately settled for a combination of the latter two approaches in our prototype. After the user captures an image, it is initially placed at a fixed distance from the camera. If there is no suitable anchor point - perhaps if the user is photographing a wide scene - then the annotation could be saved in this position. If there is a desired anchor point, the user can use our Anchor tool to reposition the image along the camera's forward axis. This moves the image forwards while maintaining its original perspective.

We also considered several approaches for visually representing the position of the camera. In the case where the image is placed a variable distance from the original camera position, it is especially helpful to have this information visible to the user. In our initial prototype, we represented the camera position by a spherical marker, and drew a line from this point to the image plane. However, in testing we found this to be an unintuitive representation, as it was unclear what the line or marker indicated, and it did not explain how the size of the image plane was calculated. In our final prototype, we instead represent the relationship between the camera position and the image plane with a pyramid. This provides a direct visual representation of the field of view of the camera.

4.4.1. Anchor Tool Our Anchor tool leverages the raycasting feature of the AR library to provide a way to anchor an image to a location in the scene. When the tool is activated, we continuously perform a raycast from a fixed location on the screen, represented by a crosshair image. The AR library takes the screen coordinates of this position and attempts to determine the z-distance (forward distance from the camera) from this point to the closest point on a surface in the scene, based on the point cloud data it has collected. This raycast is not always successful, and can fail if there is insufficient point cloud data near the desired point. We observed that this tends to occur on

flat, textureless surfaces, such as plain walls or floors. If the raycast is successful in a given update, we change the color of the crosshairs to green, and adjust the distance of the image plane to match the z-distance to the detected raycast target. When the user is satisfied with the computed position, we save this as the new image distance.

4.5. Drawing Tools

We implemented two types of drawing features: image drawing, and surface drawing. Image drawing refers to simply drawing on the flat image itself, while surface drawing provides a way to annotate surfaces in the physical scene.

4.5.1. Image Drawing The simplest type of annotation is the ability to draw freeform lines on the image itself. In the previous work, this type of tool is used to label the location of efflorescence damage on the exterior of a building. We implemented a simple drawing tool as a modular DrawComponent attached to the image plane. This component works by detecting when the user touches the image plane, and tracing the location of their movements on the surface. The component then plots the captured positions using a LineRenderer. When the user finishes drawing a line, the coordinates are returned to the AnnotationController using a OnNewLines event. After exiting this mode, the AnnotationController then saves the new line data to the server.

4.5.2. Surface Drawing A more interesting type of annotation is the ability to draw directly on surfaces in the scene, even when the surface is not parallel with the image plane. For example, the user may find it most expressive to annotate the surface of a curved wall, a task which would not be possible on a flat image. This is made possible by leveraging the AR toolkit's raycasting feature to determine the geometry of locations in the scene. Using this input, the user can draw lines directly on surfaces in the scene by moving their phone around the area to capture. We implemented this functionality in the SurfaceDrawComponent class. Similar to the DrawComponent class, this tool also works by being attached to the image plane, and exposes an event-based interface to the AnnotationController to receive captured positions. Rather than measuring the position of the user's mouse or touch input on the image plane, however, the SurfaceDrawComponent instead uses the

AR raycasting tool to locate the world-space coordinate of the position under the crosshair on the screen. Connecting together multiple coordinates in the world space, the component captures the desired outline.

Using the raycasted coordinates in world space, we can render the drawing directly on the surface of the target objects. However, in order to also make these surfaces visible on a desktop, we need to project them back onto the image plane. This transformation is performed by projecting a ray starting from the image's original camera location, through the image plane, and ending at the surface point in the scene. The intersection of this ray with the image plane gives us the coordinates needed to render a drawing directly on the image. If the image is properly positioned, then the drawing on the image plane will appear to be located at the same spot as the drawing in the scene, except flattened onto the image. This approach gives us the benefit of being able to perform 3d drawings that can also be viewed on a 2d screen.

4.5.3. Map We implemented a map view using the Mapbox SDK as a means to display the location of captured annotations. This turned out to be a less than straightforward task in Unity. Compared to a webpage, where we could have simply embedded a third-party map and used their marker tools, in Unity everything must be assembled using physical analogies. The Mapbox SDK provides a AbstractMap script which can render a map anywhere in the scene. We placed this map on the ground plane and configured it in its "World Scale" rendering mode. This causes the SDK to render the map at 1:1 scale, such that 1 meter in real-world distance on the map corresponds to 1 unit in Unity. This meant that, ignoring drift and assuming the map is properly initialized with a starting position and rotation, the AR-tracked position of the user in the scene would match up with their real-world position on the map. Since the map is in the same scene as the rest of the UI and annotations, we configured the objects to render to a separate "Map" layer which we hid from the main camera, and the configured a separate MapCamera to render only this layer. This MapCamera was then positioned above the map and aimed directly downwards in the negative Y-direction. This camera renders to a RenderTexture, which was then configured to display on Image views in the UI.

Panning and Zooming With this map representation, panning the map is performed by simply moving the camera horizontally, and zooming the map is done by moving the camera vertically. When the user drags the map, we project their mouse location to the map plane by first converting the screen location to a location within the map view, and then projecting this from the camera to the map itself. As they move the map, we calculate the distance delta in map coordinates, and apply the opposite transformation to the camera itself to create a panning effect. Pinch-to-zoom functionality is implemented in a similar manner. We project two points and calculate the ratio of the distances between the two touch points across time frames. Since the camera's horizontal view width is proportional to its vertical distance from the subject, we then apply this ratio to the Y-height of the camera to simulate zooming into or out of the map.

Map Markers We implemented markers for the map to indicate locations where annotations were added. These markers need to remain a constant size as the user zooms into and out of the map in order to remain useful as touch targets. Since the camera is moving closer to and further away from the ground plane in virtual space, we instead attached the markers to a flat rendering canvas attached to the camera at a fixed distance. We implemented this functionality in the MapCanvasController class. This class associates every currently-active AnnotationController with a marker image on this rendering canvas. On every frame update, the controller computes a new location for the marker on the canvas by projecting its 3d position from the scene to a 2d position relative to the MapCamera.

4.6. Server Architecture

Our server uses a simple event-based architecture built on top of the Socket.IO framework, using the Flask-SocketIO library implementation. In this prototype, the server provides the following functionality:

- Synchronize annotations created or updated on one device with all other connected devices.
- Save annotation changes to a local database.
- Send saved annotations to each device when it first connects to the server.
- Process incoming pose and location updates and send new alignments.

We build a local database of saved annotations, storing all objects keyed by their unique IDs. Since we use unique GUID identifiers for all model objects, we are able to use a shared primary key for all types of objects in the database, simplifying the server data storage design.

When a device first connects to the server, we can detect this connection via the built-in "connect" event. The server retrieves annotation data from the database and sends it to the client. To reduce the bandwidth of this initial load, the server does not send image data on this initial connection.

When a device sends a new or updated annotation or image, we receive this via our custom "model" event endpoint. If the annotation or image matches the GUID of an existing row in the table, then we know to update the existing item. If not, we insert a new item into the database. If the new or updated item is an annotation, we then broadcast this change to all connected clients via the Socket.IO "emit" method with the "broadcast" option set to True.

Since we do not broadcast images and require clients to manually request them as-needed, we also expose a "fetch" endpoint. After a client sends the desired image ID to this endpoint, the server responds with a copy of the desired data. In the current prototype, this endpoint is only used for fetching image data.

We process incoming pose and location updates via the "pose" and "location" events, respectively. When these updates are received, we process them via our Kalman filter, and compute an updated alignment. An in-memory dictionary is used to maintain a mapping from each client to its current alignment state.

4.7. World Alignment

In order to provide accurate localization over time, we need to combine tracking information from both the AR system and from GPS data. As we previously discussed, AR tracking works through relative positioning, and thus suffers from drift over time as the user deviates from the original location. In contrast, GPS data is absolute and does not drift, but suffers from high uncertainty at each location. To combine the best of both, we use a Kalman filter to incorporate data from each source. [26]

Kalman filters are a statistical estimation technique for predicting the true state of a underlying system using a sequence of noisy output data. It is an optimal estimator, as it is able to minimize the covariance of the predicted output. It is able to model dynamic systems which can be broken into two steps:

1. *time/process step*, where we have a way to estimate the relative change between one state and the next
2. *measurement step*, where the resulting state can be measured

Even if the predictions and measurements of these steps are noisy, the Kalman filter combines information from each updated data point to produce an estimate more accurate than any individual measurement. It outputs a prediction of the underlying state and the covariance of its estimate at each step. The Kalman filter assumes that the noise in each measurement is Gaussian, a reasonable assumption in most situations.

Formally, in matrix form, the Kalman filter models a process taking the following form:

$$x_k = Ax_{k-1} + Bu_k + w_{k-1}$$

with a measurement of the form:

$$z_k = Hx_k + v_k$$

where A represents a mapping from the previous state x to the next, B (an optional parameter) maps a process control input u_k to the resulting actual state, H maps the actual state to the measured state, and the random variables w_k and v_k model the process and measurement noise respectively. These random variables are assumed to be Gaussian with covariance Q (process covariance) and R (measurement covariance).

By solving for the minimum covariance, we get the following Kalman update equations for the process update:

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_k$$
$$\hat{P}_k^- = AP_{k-1}A^T + Q$$

and the following equations for the measurement update:

$$K_k = P_k^- H^T (HP_k^- H^T + R)^{-1}$$
$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-)$$
$$P_k = (I - K_k H)P_k^-$$

In our application, we use the AR position update as the process update step, since the AR framework determines the position of the device using a relative calculation from one frame to the next. We then use GPS location updates for the measurement update step, since GPS location is the absolute measurement we are attempting to match.

Each GPS location update provides an "accuracy" parameter which represents the precision of this measurement as determined by the strength of the received signal. According to the Android documentation, this number represents the radius of 68% confidence around the returned location. [1] If we assume that the noise in each GPS measurement is Gaussian, this size of confidence interval is equivalent to 1 standard deviation around the sample. Thus, we set the measurement covariance R_k for each k equal to the reported accuracy squared. The accuracy returned by the device is a scalar value, so we assume x and y to be independent values and set the variance to be equal along both axes.

The process variance Q is harder to estimate, as the AR framework does not return a value indicating its confidence in each position update. Based on our understanding of the system, we make the assumption that the variance of each measurement is proportional to the total distance

traveled since the previous process update step. Since variance is additive across independent samples, we approximate the total variance by calculating the total distance traveled and multiplying by a constant factor v determined empirically. We found that the model was not highly sensitive to the value of this parameter. We found that $v = 0.01$ provided good results in our experiments.

We based our Python implementation on a sample Kalman filter provided on the Scipy website.

4.7.1. Heading Estimation Since the AR-predicted position is performed relative to the starting point, it not only starts at position $(0,0,0)$, but it also starts at a heading of 0 degrees, unaligned with magnetic north. The device is able to accurately measure its tilt and yaw using its built-in accelerometer, but due to the noisiness of the compass sensor, we cannot rely on this sensor to measure the heading accurately.

Rather than relying on the compass sensor, we instead compute an estimate of the device heading by measuring the change in GPS location and comparing this with the reported AR positions in order to estimate the unknown parameter θ. Note that the prediction returned by the Kalman filter is a function of θ, since the Kalman filter runs on $< x, y >$ positions which depend on the initial choice of $\hat{\theta}$. If the value of $\hat{\theta}$ is not correct, the Kalman filter will fail to improve the accuracy of the measurement.

We can formulate the estimation of $\hat{\theta}$ as a Maximum Likelihood Estimation problem, where likelihood is measured by the conditional probability of each GPS location update:

$$\prod_k P(z_k|\hat{x}_k^-, \theta)$$

In other words, we seek to maximize, across the entire interval, the probability of observing each GPS measurement given the predicted position from the process update step. Since we model the noise in each GPS measurement as Gaussian with variance $R_k = r_k I$, maximizing this product to find $\hat{\theta}$ is equivalent to maximizing the log-likelihood:

$$\hat{\theta} = \arg\max_{\theta} \ln\left(\prod_k \frac{1}{\sqrt{2\pi r_k}} e^{\frac{(z_k - \hat{x}_k^-)^2}{2r_k}}\right)$$
$$= \arg\max_{\theta} \sum_k \left(-\frac{1}{2}\ln(r_k) - \frac{1}{r_k}(z_k - \hat{x}_k^-)^2\right)$$

We frame this as a optimization problem where we seek to minimize the negative log-likelihood over multiple runs of the Kalman filter algorithm, each initialized with a different θ. We used the `scipy.optimize.minimize_scalar` convex solver in "bounded" mode for $\hat{\theta} \in [0, 2\pi]$ to find the optimal fit.

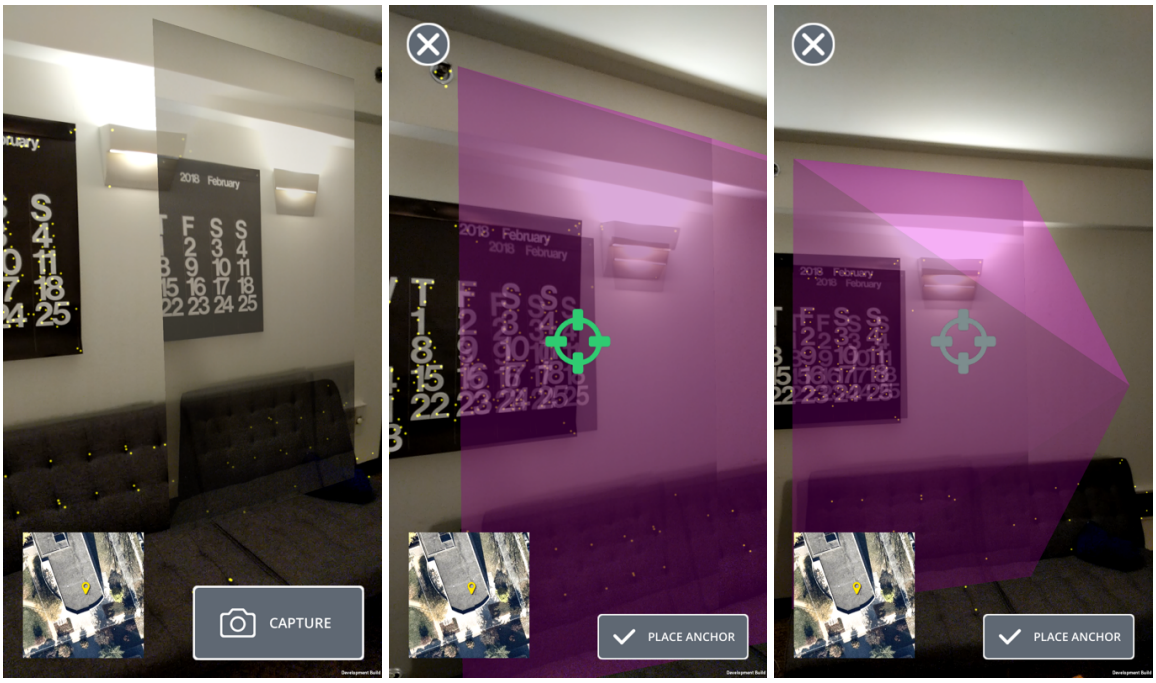
5. Evaluation

We focus our evaluation on two parts. First, we evaluate our prototype and the implemented features. We discuss the implications of our design and implementation decisions, and examine the limitations of the AR framework. Finally, we evaluate the performance of the world alignment algorithm and discuss the implications of the results on the performance of AR tracking.

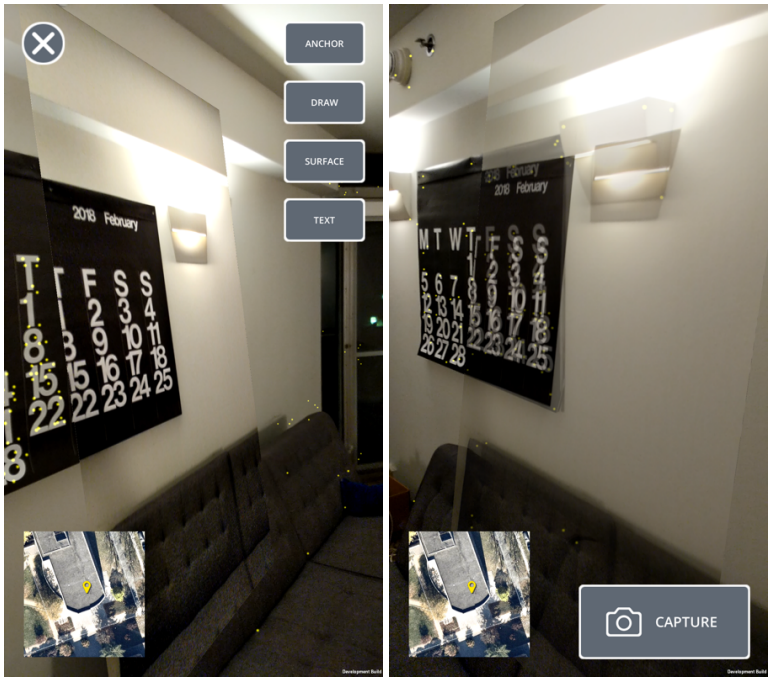
5.1. Prototype

The final prototype created for this paper is shown below in Figure 1. The app opens directly to the camera view and presents the user with a map preview and a capture button. As previously described, pressing this capture button takes a photo using the camera and inserts it into the scene at a fixed distance from the camera position. By moving around, we can see how the image plane remains in place.

Tapping the image opens the annotation editing tools. Using the anchor tool, the user can move the image plane to affix it to an object or surface. Figure 1 shows the usefulness of this approach. At first, since the captured image is located a distance away from the wall, we observe a strong parallax effect as we move around in the scene. Using the anchor tool, we can select a spot on the wall to attach the image plane. We also see how the image pyramid provides a visual hint for

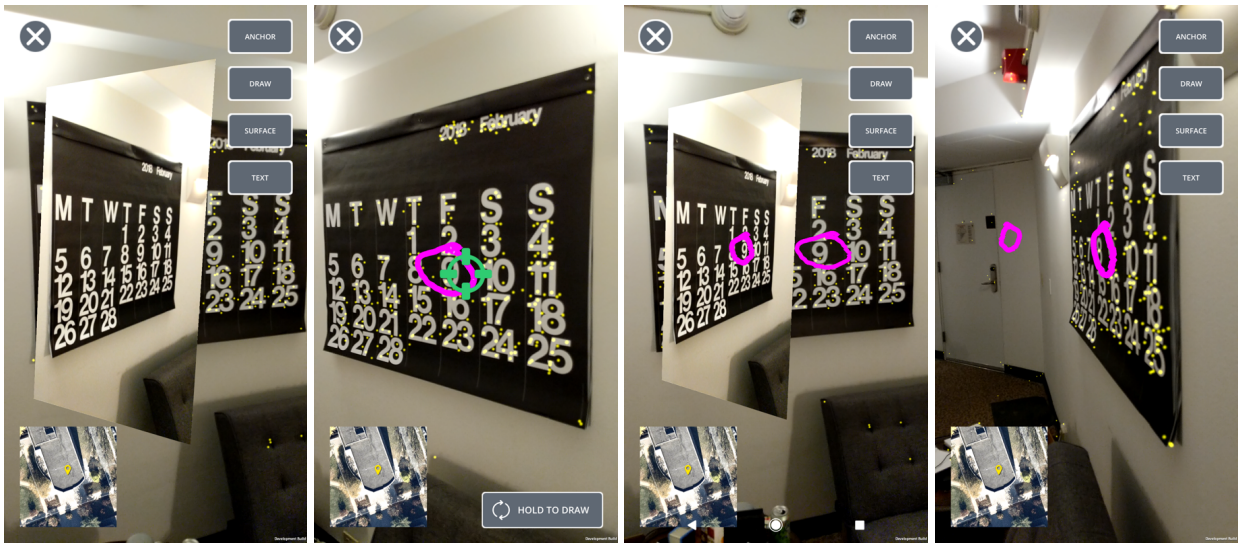


(a) Offset from camera. (b) Placing on wall. (c) Showing pyramid.



(d) Side view. (e) Another side view.

Figure 1: Anchor tool demonstration.



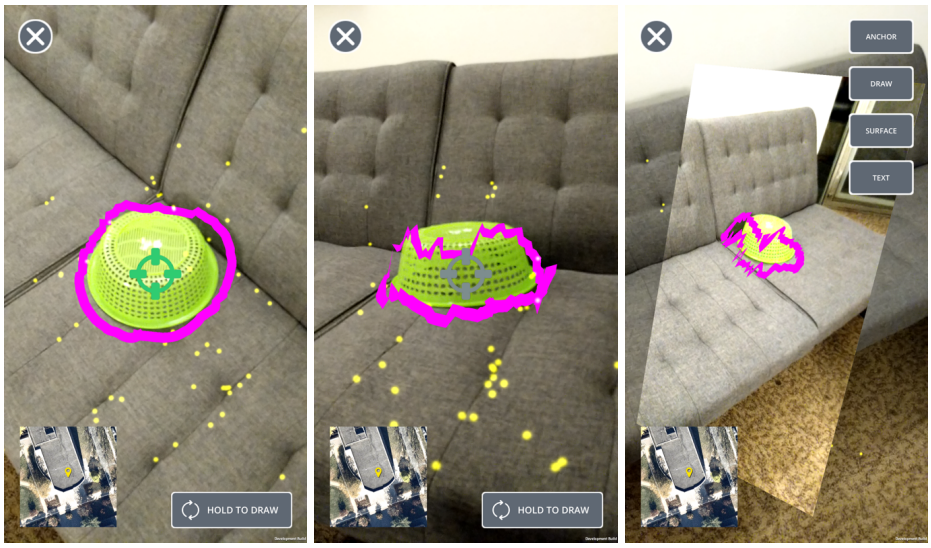
(a) (b) (c) (d)

Figure 2: Surface drawing tool demonstration.

how this sizing is a result of the camera's perspective. After anchoring the annotation, we see how it now aligns with the wall and can now be viewed from multiple angles and positions. Unlike a panorama-based approach, we observe how our image-plane approach has this benefit of being able to align flat against a desired surface.

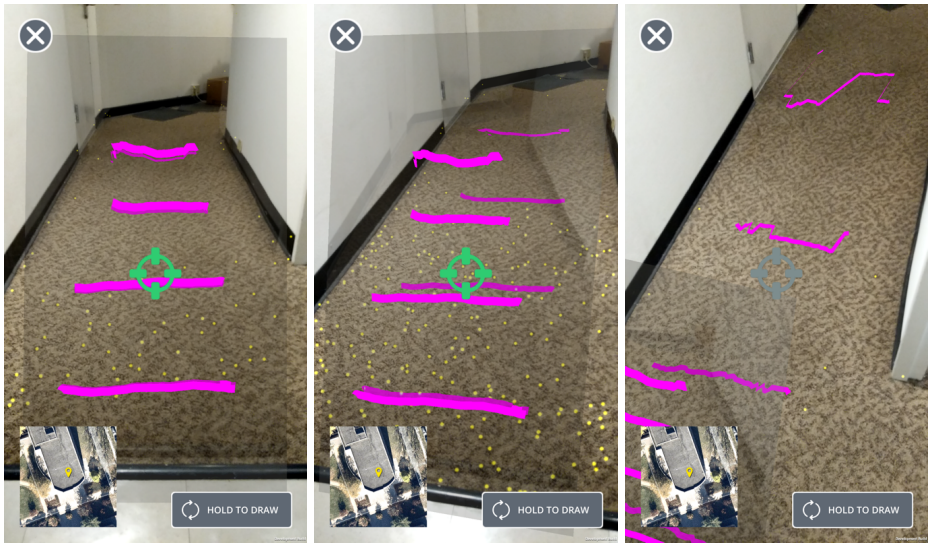
Next, we investigate the surface drawing tool. In a best-case scenario shown in Figure 2, where the surface is flat and uniformly patterned, the tool performs well. We see how the AR framework is able to reliably detect the surface, as shown by the yellow dots indicating the point cloud it detected. Even though we captured the initial image from an angle to the left of the wall, we are still able to draw flat on the wall itself and view this drawing from multiple angles. Returning back to the original image, we see how it is able to accurately reproject the drawing back onto the image plane for later viewing on a computer.

However, we observe that the surface drawing tool runs into problems against more complex backgrounds or at more extreme angles. In Figure 3, we see how a more complex geometry confuses the AR framework's raycasting feature. Though the drawing appears normal from the original angle it was made, viewing it from other angles shows significant noise in the z-direction. We also see how the tool is not able to perform visual occlusion behind objects in the scene, since it is only



(a) (b) (c)

Figure 3: Surface drawing tool issues.

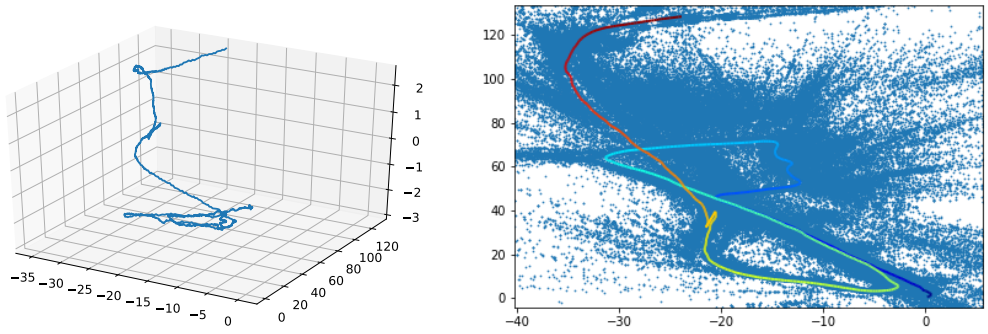


(a) (b) (c)

Figure 4: Surface drawing tool issues at sharp angles.

measuring distance to a point cloud and does not have knowledge of the true geometry.

Figure 4 examines this issue in additional detail. In this case, we captured an image at a low angle to the ground, and then drew several horizontal lines from this perspective. On the original image, these lines appear straight, but after moving upwards to view the drawing from a higher perspective, we again see how z-distance uncertainty from the camera leads to significant noise in the detected surface.



(a) 3D View (b) Point Cloud

Figure 5: Test 1 path and point cloud data.

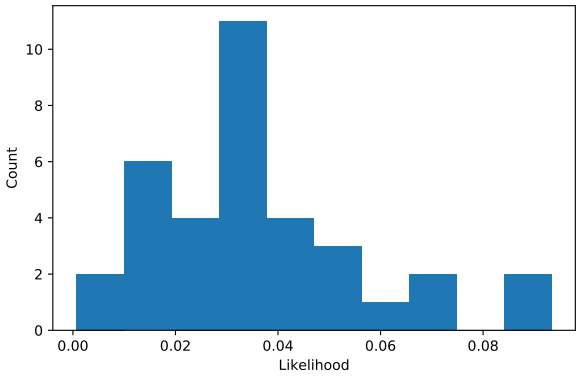


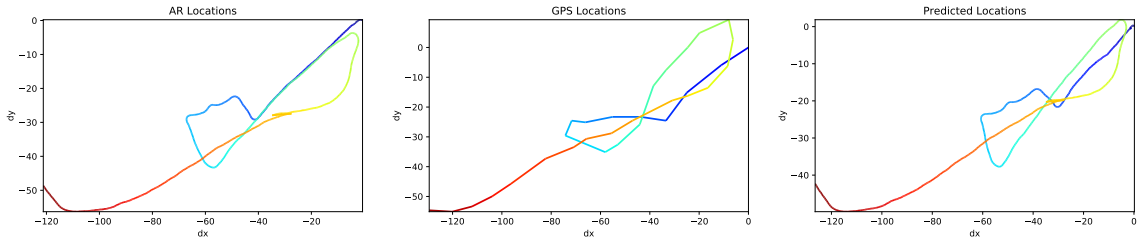
Figure 6: Histogram of likelihood values for each location sample in test 1.

5.2. World Alignment

To evaluate the performance of the world alignment algorithm, we performed several tests at Streicker Bridge. These tests were performed during the daytime under clear skies.

5.2.1. Test 1 In the first test, we start from the east, walk beneath the bridge, cross a road, then return to the east end of the bridge. We then cross the bridge to the west and exit to the north towards the adjacent buildings. Figure 5 visualizes a 3d plot of the AR-tracked path, showing how the AR library is aware that we walked at different altitudes. The adjacent figure also shows the AR path overlaid over the detected point cloud. This path is shown before either the heading alignment procedure or location prediction steps are performed.

Our heading alignment algorithm returns an estimate $\hat{\theta} = 101.3°$. We visualize the computed likelihood at each location update using the histogram in figure 6. The x-axis shows the calculated



(a) AR Locations (b) GPS Locations (c) Predicted Locations

Figure 7: Test 1 Kalman filter output.

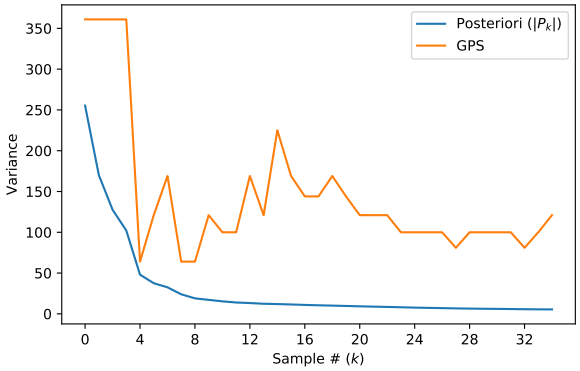


Figure 8: Test 1 predicted variance decreasing over time.

likelihood values, with higher values indicating better fit.

Applying this transformation to the AR estimates results in the plot in figure 7, which shows the locations returned by both AR and GPS positioning. This figure also shows the final aligned result.

We see that the AR-tracked position performs well at tracking when a user returns back to the same location they were before. When we walked back along the road to return to the end of the bridge, the AR position accurately retraces the original path. In comparison, the GPS locations wander significantly, and it is not clear from the GPS trace that we did indeed trace the same path. We also see that during the time we are under the bridge, which is the loop in the middle, GPS accuracy and update rate is low. This shows a significant advantage of AR tracking, as it is able to continue tracking even when there is no clear view of the sky. In fact, it likely performs better in these enclosed environments, as it would be able to track more nearby surfaces.

The final predicted output appears skewed as a result of the inaccurate GPS data, especially during the loop under the bridge. At first glance, it may appear that the predicted positions, as reported by the Kalman filter, look less accurate for parts of the path. However, this behavior is an artifact of

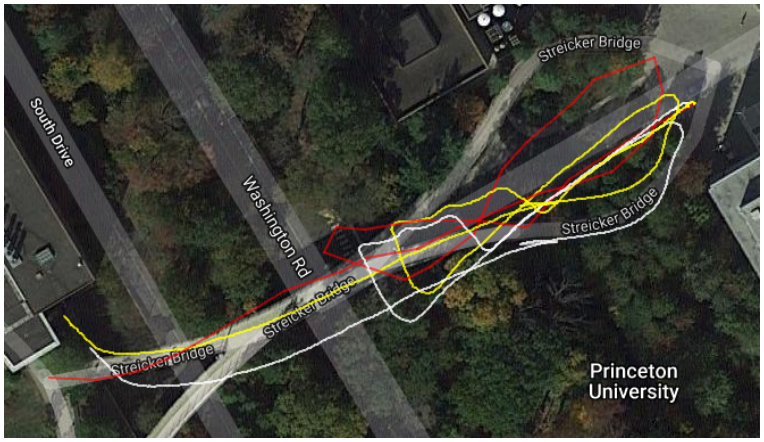


Figure 9: Test 1 output shown on a map. White lines = AR positions, red lines = GPS positions, yellow lines = predicted output.

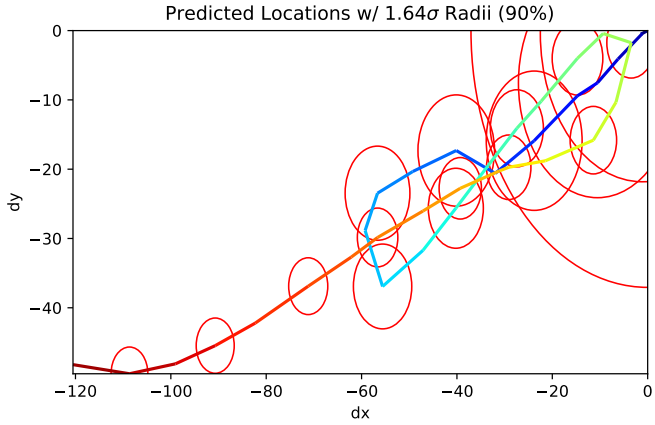
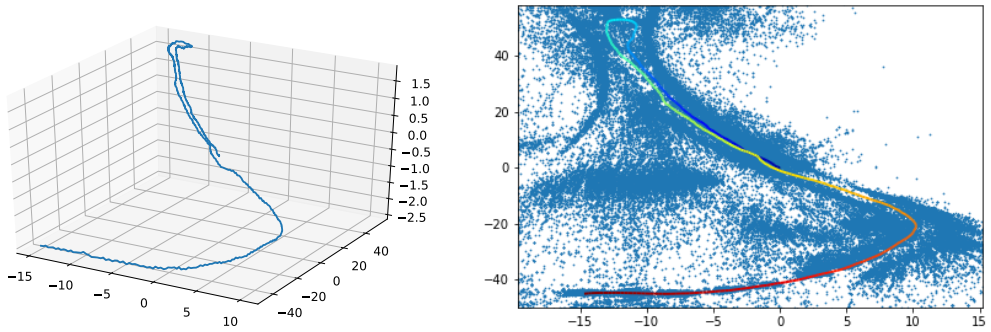


Figure 10: Test 1 predicted locations with their associated uncertainty.

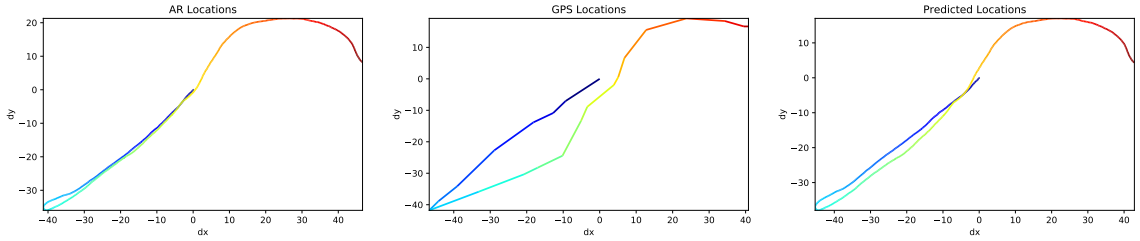
the filtering algorithm. Since the Kalman filter is a recursive smoothing algorithm, it improves the accuracy over time. Figure 8 shows how the computed variance $|P_k|$ (in blue) decreases over time as we approach the end of the test, and how it is able to improve significantly over the variance of individual GPS locations (in orange). When we look at the result on a map in Figure 9, we see how the accuracy actually converges towards the correct location over time as we approach the end of the path.

In Figure 10, we visualize the uncertainty in the location prediction by plotting circles along the path to represent the predicted variance P_k. (Note that in the visualization, these circles appear as ovals since the scale of the two axes differ.) Each circle has a radius equal to $1.64\sigma = 1.64\sqrt{|P_k|}$, approximating a 90% confidence interval around the prediction. We see how the variance is very



(a) 3D View (b) Point Cloud

Figure 11: Test 2 path and point cloud data.



(a) AR Locations (b) GPS Locations (c) Predicted Locations

Figure 12: Test 2 Kalman filter output.

high at the start of the path, but as we continue along the bridge, the variance drops significantly. This helps to explain the converging behavior we observed.

5.2.2. Test 2 In the second test, we start from the east end of the bridge, walk along the bridge to the center, and then return along the bridge before exiting and turning south. Figures 11, 12, and 13 show the results of this test. We see that the AR location again vastly outperforms GPS at retracing

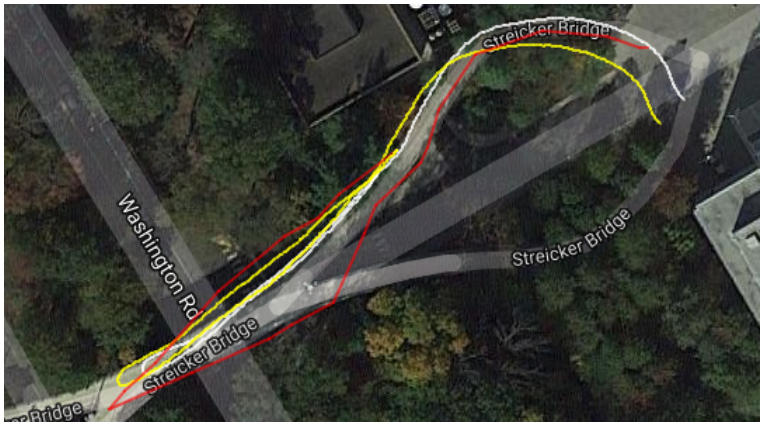
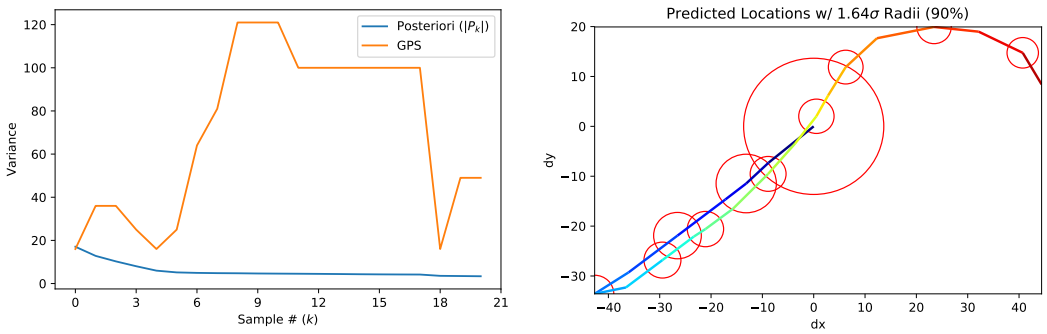
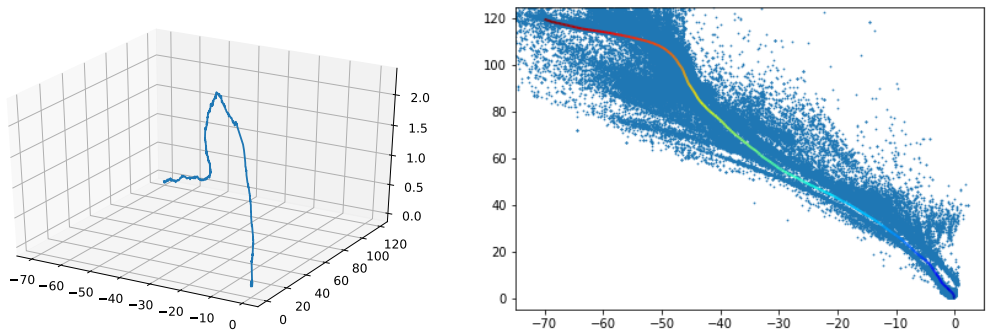


Figure 13: Test 2 output shown on a map.



(a) Variance vs. Time (b) Locations w/ Variance

Figure 14: Test 2 predicted variance decreasing over time.



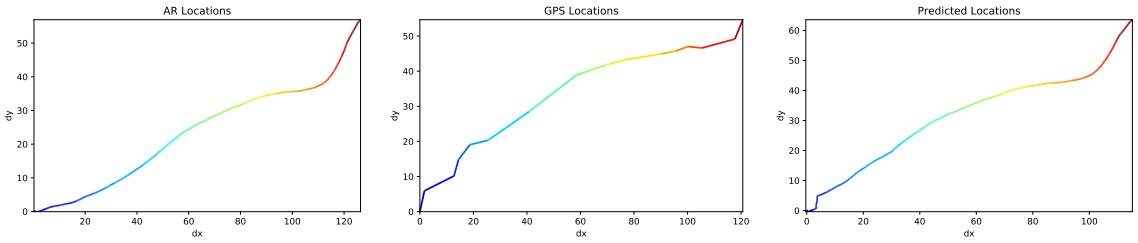
(a) 3D View (b) Point Cloud

Figure 15: Test 3 path and point cloud data.

steps along the bridge. In this case, it appears that the GPS may have lost signal at some point, since there are few points along the green, middle portion of the test, yet there are several points along the starting blue portion. Regardless, the predicted location performs well in this case, though we see a strange lagging effect near the western end of the bridge. At this point in the test, both the AR and GPS locations are accurate, but the predicted location is further away from either of them. This may be due to the predicted location being shifted further to the west near the turn at the center of the bridge.

Again, we can visualize the decrease in uncertainty over time in figures 14. We see how the Kalman filter output is able to reduce the variance of the predicted output even when GPS variance rises significantly near the middle of the route.

5.2.3. Test 3 In the third test, we walk straight across the bridge from the northwest entrance to the southeast exit. Figures 15, 16, and 17 show the predicted results, and figure 18 shows how variance



(a) AR Locations (b) GPS Locations (c) Predicted Locations

Figure 16: Test 3 Kalman filter output.

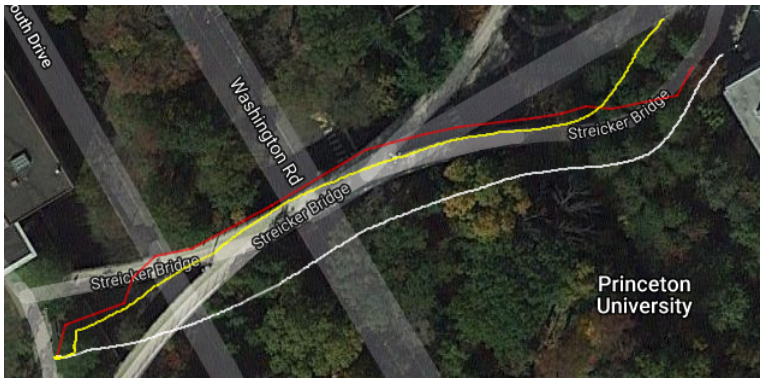
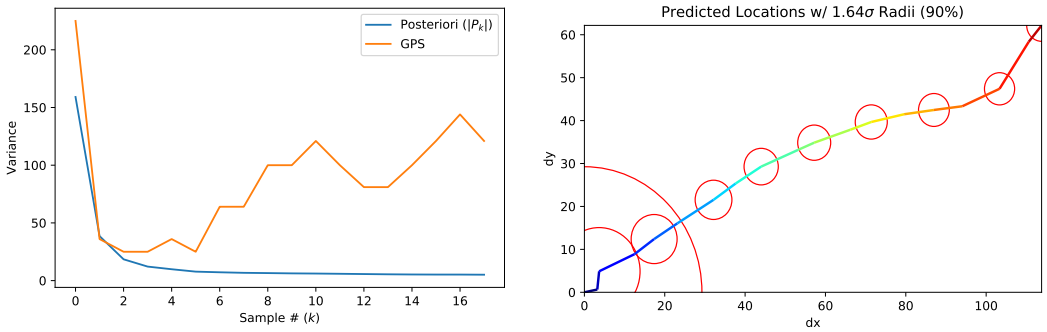


Figure 17: Test 3 output shown on a map.

changes over the path. We see an interesting outcome in this experiment where the GPS data quickly pulls the AR location back to the correct path, but as we approach the end, the path mysteriously deviates from the GPS line, even though the AR path correctly follows the shape of the bridge exit (despite being offset), and the GPS data is actually accurately tracking the bridge. One possible explanation for this behavior is that the GPS data may be lagging behind in time. Another theory is that the initial heading of the path may be off, causing the predicted path to skew too far north.



(a) Variance vs. Time (b) Locations w/ Variance

Figure 18: Test 3 predicted variance decreasing over time.

6. Conclusions

In this paper, we developed and evaluated a framework for performing cultural heritage documentation in augmented reality. We designed and implemented a cross-platform, client-server system for creating, saving, and viewing annotations. We described the implementation ideas and pitfalls behind several of our tools, and evaluated the practicality and performance of our system. Our findings show the promise of augmented reality as a useful framework for future development in the field of cultural heritage.

7. Future Work

7.1. Relocalization

Although we are able to determine a user's position to a reasonable degree of precision, our localization approach is still limited in its ability to accurately relocalize a user across sessions. GPS data is subject to both random noise and systematic error, so the resulting alignment offset from session may be insufficiently accurate to match the location result of a second session, a problem called relocalization. There are several techniques to explore in this area, which lie beyond the scope of this paper.

One promising algorithmic approach is to leverage the point cloud data we collect in order to build a rough server-side 3d model of the scene. When a separate session is started, the point cloud data from this second session can then be compared with the one from the first in order to determine their offset. Techniques designed for point cloud registration, the task of computing the offset between two point clouds of the same object, could prove successful in this area.

We briefly investigated the feasibility of the Coherent Point Drift algorithm, a probabilistic method of point cloud alignment which is robust to outliers and noise. This algorithm works by alternately computing the match probability of pairs of points, and running a minimization step to iteratively compute the most likely transformation. We found that the algorithm would suffer from global alignment issues, where it would tend to align two point clouds incorrectly if one was a

significantly smaller subset of the other. Further research in this area would be needed to investigate this and other algorithms, such as the Iterative Closest Point algorithm.

Another approach is to give the user the task of finding the proper alignment by framing the issue as a UI problem. A user flow could be developed to guide the user to realign themselves with a previous session by matching the locations of key points in the physical area. For example, using annotations from the prior session, they could be asked to move their device to their original positions and rotations by matching one or more annotation images with the camera. The results from multiple such alignments can then be triangulated to compute an overall alignment.

7.2. Further Development

Once the relocalization problem and other issues discussed in this paper are addressed, a full end-user prototype can be built for user testing and evaluation. This prototype should include functionality for signing in as a particular user, a feature requested in our user statements. It should also contain help text and tutorials in order to instruct the user how to use the tool. The system should also provide a mechanism for selecting a current project, so that annotation data for unrelated projects or buildings is not kept together in the same scene, cluttering the interface.

8. Acknowledgments

I would like to thank my advisors, Professor Branko Glisic and Rebecca Napolitano, for the inspiration and motivation behind this project and their support of this work throughout the semester. Special thanks as well to Rebecca for helping refine the ideas that went into designing the final prototype and its set of features.

9. Honor Code

This paper represents my own work in accordance with University regulations. — Zachary Liu

References

[1] "Location." [Online]. Available: https://developer.android.com/reference/android/location/Location

[2] "rawFeaturePoints - ARFrame." [Online]. Available: https://developer.apple.com/documentation/arkit/arframe/2887449-rawfeaturepoints

[3] J. Atwood, "Understanding Model-View-Controller," 2018. [Online]. Available: https://blog.codinghorror.com/understanding-model-view-controller/

[4] S. Gardonio, "ARCore vs. ARKit: Google Counters Apple," 2017. [Online]. Available: https://www.iotforall.com/arcore-vs-arkit-comparison/

[5] B. Glisic, D. Inaudi, and N. Casanova, "SHM process as perceived through 350 projects," K. J. Peters, W. Ecke, and T. E. Matikas, Eds., vol. 7648. International Society for Optics and Photonics, mar 2010, p. 76480P. [Online]. Available: http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.852340

[6] B. Glisic, M. T. Yarnold, F. L. Moon, and A. E. Aktan, "Advanced Visualization and Accessibility to Heterogeneous Monitoring Data," *Computer-Aided Civil and Infrastructure Engineering*, vol. 29, no. 5, pp. 382–398, may 2014. [Online]. Available: http://doi.wiley.com/10.1111/mice.12060

[7] F. Hassani, "Documentation of cultural heritage techniques, potentials and constraints," in *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences - ISPRS Archives*, vol. 40, no. 5W7, 2015, pp. 207–214. [Online]. Available: https://pdfs.semanticscholar.org/5a63/5c77be1250450824135b396890b305ac59f1.pdf

[8] H. Kim, G. Reitmayr, and W. Woo, "IMAF: In situ indoor modeling and annotation framework on mobile phones," *Personal and Ubiquitous Computing*, vol. 17, no. 3, pp. 571–582, 2013.

[9] M. Kourogi, T. Kurata, K. Sakaue, and Y. Muraoka, "Improvement of panorama-based annotation overlay using omnidirectional vision and inertial sensors," *Digest of Papers. Fourth International Symposium on Wearable Computers*, pp. 3–4, 2000.

[10] Y. Kourogi, M.; Sakaue, K.; Muraoka, "A panorama-based technique for annotation overlay and its real-time implementation," *Multimedia and Expo, 2000. ICME 2000. 2000 IEEE International Conference on*, no. September, pp. 657–660, 2000. [Online]. Available: http://scholar.google.com/scholar?hl=en{&}btnG=Search{&}q=intitle:No+Title{#}0

[11] T. Langlotz, C. Degendorfer, A. Mulloni, G. Schall, G. Reitmayr, and D. Schmalstieg, "Robust detection and tracking of annotations for outdoor augmented reality browsing," *Computers & Graphics*, vol. 35, no. 4, pp. 831–840, aug 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0097849311001075

[12] T. Langlotz, S. Mooslechner, S. Zollmann, C. Degendorfer, G. Reitmayr, and D. Schmalstieg, "Sketching up the world: In situ authoring for mobile Augmented Reality," *Personal and Ubiquitous Computing*, vol. 16, no. 6, pp. 623–630, aug 2012. [Online]. Available: http://link.springer.com/10.1007/s00779-011-0430-0

[13] T. Langlotz, D. Wagner, A. Mulloni, and D. Schmalstieg, "Online creation of panoramic augmented reality annotations on mobile phones," *IEEE Pervasive Computing*, vol. 11, no. 2, pp. 56–63, feb 2012. [Online]. Available: http://ieeexplore.ieee.org/document/5551109/

[14] H. Lee, J. Lee, S. Ahn, H. Ko, and G. Kim, "Collaborative AR Authoring Framework using Remote Panorama and In-situ Sensors," *Imrc.Kist.Re.Kr*, no. April, 2014. [Online]. Available: http://www.imrc.kist.re.kr/MediaWiki/images/9/98/Lee.pdf

[15] R. Letellier, *Recording and information management for the conservation of heritage places: guiding principles*, 2007, vol. Guiding pr. [Online]. Available: https://www.getty.edu/conservation/publications{_}resources/pdf{_}publications/pdf/guiding{_}principles.pdf

[16] L. Marques, J. António Tenedório, M. Burns, T. Romão, F. Birra, J. Marques, A. Pires, and J. António, "CULTURAL HERITAGE 3D MODELLING AND VISUALISATION WITHIN AN AUGMENTED REALITY ENVIRONMENT, BASED ON GEOGRAPHIC INFORMATION TECHNOLOGIES AND MOBILE PLATFORMS." [Online]. Available: https://upcommons.upc.edu/bitstream/handle/2117/101733/4686-2123-2-PB.pdf

[17] A. Myronenko and X. Song, "Point Set Registration: Coherent Point Drift," 2009. [Online]. Available: https://arxiv.org/pdf/0905.2635.pdf

[18] R. Napolitano, A. Blyth, and B. Glisic, "Virtual Environments for Structural Health Monitoring," *Sensors*, vol. 18, no. 1, pp. 1549–1555, jan 2018. [Online]. Available: http://www.mdpi.com/1424-8220/18/1/243

[19] R. K. Napolitano, G. Scherer, and B. Glisic, "Virtual tours and informational modeling for conservation of cultural heritage sites," *Journal of Cultural Heritage*, vol. 29, pp. 123–129, jan 2018. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S129620741730420X

[20] A. Olarnyk, "Designing for Augmented Reality – Prototypr," 2018. [Online]. Available: https://blog.prototypr.io/designing-for-ar-b276c8251c20

[21] S. Roweis and Z. Ghahramani, "A Unifying Review of Linear Gaussian Models," *Neural Computation*, vol. 11, no. 2, pp. 305–345, 1999. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.314.2260{&}rep=rep1{&}type=pdfhttp://www.mitpressjournals.org/doi/10.1162/089976699300016674

[22] D. W. F. Van Krevelen and R. Poelman, "A Survey of Augmented Reality Technologies, Applications and Limitations," 2010. [Online]. Available: https://www.researchgate.net/profile/Rick{_}Van{_}Krevelen2/publication/279867852{_}A{_}Survey{_}of{_}Augmented{_}Reality{_}Technologies{_}Applications{_}and{_}Limitations/links/58dab7f445851578dfcac285/A-Survey-of-Augmented-Reality-Technologies-Applications-and-Limitations.pd

[23] V. Vlahakis, N. Ioannidis, J. Karigiannis, M. Tsotros, M. Gounaris, L. Almeida, D. Stricker, T. Gleue, I. T. Christou, and R. Carlucci, "Archeoguide," *Proceedings of the 2001 conference on Virtual reality, archeology, and cultural heritage - VAST '01*, no. January, p. 131, 2001. [Online]. Available: http://portal.acm.org/citation.cfm?doid=584993.585015

[24] D. Wagner, A. Mulloni, T. Langlotz, and D. Schmalstieg, "Real-time panoramic mapping and tracking on mobile phones," in *2010 IEEE Virtual Reality Conference (VR)*. IEEE, mar 2010, pp. 211–218. [Online]. Available: http://ieeexplore.ieee.org/document/5444786/

[25] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, and D. Schmalstieg, "Real-time detection and tracking for augmented reality on mobile phones," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 3, pp. 355–368, may 2010. [Online]. Available: http://ieeexplore.ieee.org/document/5226627/

[26] G. Welch and G. Bishop, "An Introduction to the Kalman Filter," *In Practice*, vol. 7, no. 1, pp. 1–16, 2006. [Online]. Available: http://www.cs.unc.edu/{~}{%}7Bwelch,http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.79.6578{&}rep=rep1{&}type=pdf

[27] H. M. Yilmaz, M. Yakar, S. A. Gulec, and O. N. Dulgerler, "Importance of digital close-range photogrammetry in documentation of cultural heritage," *Journal of Cultural Heritage*, vol. 8, no. 4, pp. 428–433, sep 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1296207407001094

[28] P. A. Zandbergen, "Accuracy of iPhone Locations: A Comparison of Assisted GPS, WiFi and Cellular Positioning," *Transactions in GIS*, vol. 13, pp. 5–25, jun 2009. [Online]. Available: http://doi.wiley.com/10.1111/j.1467-9671.2009.01152.x