Indirect Augmented Reality Browser for GIS Data

Patrick Skinner*
University of Otago

Jonathan Ventura†
University of Colorado
Colorado Springs

Stefanie Zollmann‡
University of Otago

Figure 1: Indirect Augmented Reality used for exploring geographic data in urban settings. Left and Middle) Labels aligned with building faces. Right) Visualisation of point features using a shadow highlight on the ground.

Abstract

In Augmented Reality applications, user experience is highly dependent on the accuracy of registration between digital content and the real world. Errors in tracking and registration can arise due to inaccuracy of sensors or challenging conditions such as urban canyon effects or magnetic distortions. Indirect augmented reality is an approach that avoid these issues by using precaptured and preregistered images instead of a live video feed. However, indirect augmented reality highly depends on the availability of those preregistered images. In particular, when being used for browsing geographic information, it is important to access data in an omnipresent way.

In this work, we propose an Indirect Augmented Reality browser that aims to address these availability problems by combining indirect Augmented Reality with crowd sourced precaptured street level imagery with geospatial data. We demonstrate how our indirect augmented reality browser annotates buildings and landmarks in the users' environment and investigate the feasibility by analysing the performance of such an approach. In addition, we investigate issues of visibility and legibility when labelling the environment.

Index Terms: Human-centered computing—Visualization—Visualization application domains—Geographic visualization; Human-centered computing—Human computer interaction (HCI)— Interaction paradigms—Mixed / augmented reality

1 Introduction

The quality of Augmented Reality (AR) applications is reliant on the accuracy of alignment between real and virtual content. The quality of sensors found in modern mobile devices is often not sufficient when trying to register virtual content with a live video feed. In particular, external influences such as the urban canyon effect can create large offsets when estimating the position in urban environments, also large urban structures can influence the accuracy of magnetic sensors. Indirect Augmented Reality avoids these issues by aligning the virtual content with precaptured images [12]. While the real world content is no longer live the increase in registration accuracy can provide an overall better experience for the user.

*e-mail: patrickskinner@outlook.com
†e-mail: jventura@uccs.edu
‡e-mail: stefanie.zollmann@otago.ac.nz

However, the advantages of Indirect AR come with a set of prerequisites. The main challenge is that preregistered images from a street level perspective have to be available close to the user's position in order to provide a good user experience [12]. This is in particular a challenge when using IAR to explore geographic information that is available in an omnipresent way driven by the growth of public domain geographic information systems (GIS).

While previous work investigated the user experience of Indirect AR systems [12] in particular in terms of how well users can localise themselves, so far there is no work on how to provide Indirect AR in an omnipresent way. In this work we address this gap by combining crowd sourced street view data and open source GIS data in an Indirect AR browser (IARB) for exploring GIS data.The proposed system allows us to create situated visualisation for all locations where crowd sourced street view data is available and making use of Indirect AR to overlay panoramic images with 3D annotations labelling buildings and points of interest in the scene. Situated Visualisation is a term first used by White and Feiner [11] to describe an AR visualisation that is both "related to and displayed in it's environment". In our application we want to provide annotation of the user's surrounding environment to aid pedestrian navigation and sightseeing. Our application is browser based and designed to run on modern mobile phones and tablets that support WebGL. The use of these openly accessible and editable data sources also provides us the benefit of being easily able to add and modify both real and virtual content for use in our system. Kurkovsky et al. [6] noted that one of the issues with augmented reality systems of this kind was the difficulty of modifying and adding new virtual content. This is due to the fact that the virtual content often needs to be created and tailored specifically for the real environment.

2 Background

The performance and user experience of Indirect AR in comparison to traditional AR has been the topic of several research papers, focusing on the viability of Indirect AR in comparison to traditional AR and the shortcomings specific to Indirect AR.

Registration Accuracy

Indirect AR is used to provide an increase in registration accuracy in comparison to traditional AR using a live video feed. Wither et al. [12] quantified the effect of orientation when annotating buildings in panoramas. Alignment with an object at 20 meters from the camera will be out by 3.75 meters with 10 degrees of orientation error, which they claim is within expected error for the sensors found on mobile devices. With indirect AR this orientation error can be removed almost entirely, depending on the quality of the images and the accuracy of the data we have about

the image and the real world environment. Traditional AR can also come with the problem of sensor data lagging behind the updating video feed, leading to misalignment and jitter as the user rotates the camera. This problem is avoided by the use of indirect AR, since the real and virtual content is aligned once and does not need to be realigned as the user pans the camera around the scene.

Spatial Inconsistencies One of the main problems with Indirect AR is the difference between the user's location and the location of the camera in the image they are shown. Wither et al. [12] tested users' ability to locate a highlighted building when given an image of that building from a different location, to test the effect of these spatial inconsistencies on user experience. They ran these tests in a variety of conditions, the main variation being the distance of the camera to users location, and the angle of the camera to the buildings facade. They found that increasing the distance between the image and the user increased the time it took for the user to locate the building in a linear fashion. In our work the distance between the user and the centre of our virtual scene is dependent on the density of images in an area. In order to provide Indirect AR in a large variety of places we decided to use he crowd-sourced street level imaging service Mapillary[1]. However spatial inconsistencies can be caused by the time taken to update our virtual scene in response to changes in the user's real world position. The time taken to process and construct a new virtual scene needs to be minimised to reduce these inconsistencies.

Temporal Inconsistencies Okura et al. [9] studied how the user experience was effected by the difference in time between their use of indirect AR and when the image they are displayed was captured. The changes in the environment caused by the passing of time are referred to as temporal inconsistencies. These inconsistencies occur at a variety of timescales. The smallest timescale is the issue of moving objects in a scene, such as the presence and position of people in the real world and the image. The next timescale is the issue of varying time of day. Lighting and weather conditions can vary even by the hour, creating discrepancies between the image and the real world. The largest timescale is variations in season, the same area can appear vastly different between summer and winter.

Okura et al. [9] conducted a user study to determine how these different inconsistencies affect the experience for the end user. Their study found that temporal inconsistencies have a greater impact on the user experience than spatial inconsistencies. Using crowd-source image data sources allows for the continuous update of images if there are enough users contributing to the database. However, it comes with the disadvantage that we have no control over the temporal inconsistencies within the data.

View Management The placement of information for clarity and legibility is one of the problems that needs to be addressed when visualising geographic data within an Indirect AR browser. View management is the layout and representation of digital information to ensure visibility and legibility. There have been many papers discussing methods of ensuring visibility when labelling objects in augmented reality. Shibata et al. [10] proposed methods of addressing overlapping labels attached to objects, using a priority system to decide which of the labels should be moved to a new location. They also discussed the repositioning of labels that were partially out of frame. The techniques discussed in this paper were designed for simple environments with few objects and would not necessarily be applicable to a large outdoor scene due to the reliance on accurate knowledge of the scene geometry.

Computer vision techniques can be used to make up for the lack of knowledge about the real environment. Grasset et al. [4] used an image-based approach for the placement of 2D annotations. This approach combined edge detection with a saliency map to create a map of the most important parts of the image. Once this map is created their "layout solver" can then position the labels in away that avoids occluding these important sections of the image while also maintaining a clear relationship between the annotation and the corresponding real world object.

[1]https://www.mapillary.com/

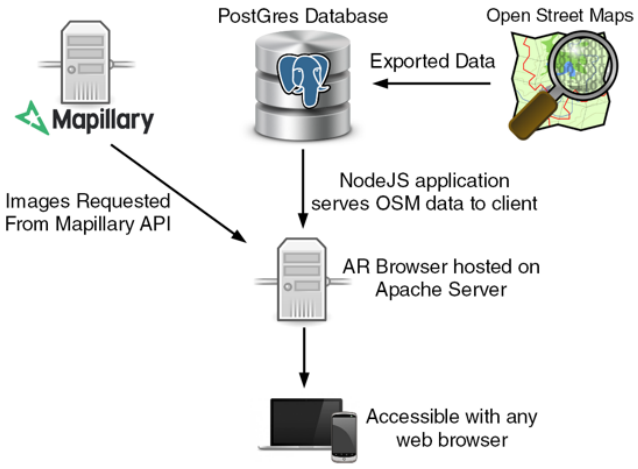

Figure 2: Overview of IARB infrastructure.

Bell et al. [1] devised a view management method for annotating buildings with in a fully 3D environment. However their method involves using a 2D projection of the 3D scene to position the labels, rather than positioning the label in 3D space. Image based techniques can be used to retrieve geometric information from an image which can then be used to position labels in relation to the world geometry. Langlotz et al. [7] used computer vision techniques to create a surface map from an image, allowing them to align labels with suitable planes in the scene.

In our Indirect AR browser, view management is done using a corse 3D representation of the scene. We use this 3D representation to determine which buildings are within the users view and select the face of each visible building that is most suitable for displaying a label. This is an extension of the work presented by Zollmann et al. [14] to address presentation issues such as the visibility and information clutter when annotating environments in AR. They used GIS data from OpenStreetMap to create a 3D scene and used the information provided by this scene to implement dynamic label placement and occlusion culling.

3 Implementation

Our approach integrates crowd-sourced street level imagery and GIS data with an indirect AR browser. For this purpose, we implemented a client-server infrastructure. As shown in Figure 2, we run our web-based indirect AR browser on the client side. The indirect AR browser accesses data from two different data sources, 1) crowd-sourced panoramic street level imagery from the Mapillary API and 2) GIS data served by a server application.

3.1 Web-based indirect AR browser

We implemented the Indirect AR browser as a browser based Javascript application using the Three.js[2] graphics library to display 3D WebGL[3] content. By using WebGL we provide platform independence and provide an application that is compatible with any modern web browser on desktop or mobile.

The basis of our Indirect AR browser is the rendering of panoramic images in the user's perspective. For this purpose, we

[2]https://threejs.org/

[3]https://www.khronos.org/webgl/

use an equirectangular projection to map the 360 degree photo to the inside of a sphere. We place a camera at the centre of the sphere and allow the user to rotate it freely with mouse or touchscreen input, allowing the user to pan freely around the entire image.

The street-level images that we use are retrieved from Mapillary, which is an openly accessible street level imaging service. Users can capture and upload image data to Mapillary by connecting a 360 degree camera to their smartphone or tablet and using Mapillary's iOS/Android application. Mapillary offered a wide variety of existing images for us to test our application with, as well as allowing us to easily upload our own images for testing in a local environment.

Mapillary provides access to their images through their API. To query image data, we make a query to their API passing a latitude/longitude pair, a radius, and the maximum number of image features we want to retrieve. The API request returns a set of image features. Each image feature is a JSON object that contains some metadata about the image and the unique key which can be used to retrieve the image file itself. When we create the Indirect AR content for a certain location, we request the 10 closest images to our location and retrieve the closest image that is tagged as a panorama in the metadata.

The client-side web application is hosted on a local Apache server for development, and deployed through Gitlab for testing on other devices.

3.2 Backend Infrastructure

The web application is supported by a series of backing infrastructure. We store the 2D map information provided by Open Street Map (OSM) in a Postgres[4] database. OSM allows developers to export their map data for their own local storage and use. We use the GIS software QGIS[5] to convert the exported data into shapefiles, a standard vector formation for geographic information objects. These shapefiles represent buildings as polygons with a set of associated metadata fields. Postgres includes tools to convert these shapefiles into PSQL objects to be stored in our database.

The database makes use of PostGIS[6], an open source extension that provides support for geospatial objects. PostGIS provides functionality that allows us to query for objects based on their location. This allows us to retrieve a subset of our map data by querying for objects within a set radius of a given coordinate pair, allowing us to retrieve the buildings surrounding the user.

The browser has access to the database through a server side Node.js application. The Node.js application accepts HTTP requests from the application and converts them to PSQL queries, the results of these queries are returned to the application as JSON objects. The objects stored in our database are using the World Geodetic System 1984 (WGS84) coordinate system, which is a global coordinate system with the origin at the Earths centre of mass. We convert these coordinates to a local coordinate system (East, North, Up) to be used in our Indirect AR scene.

The database and Node.js application are running on a local server.

4 Data Processing

The data served by the backend infrastructure requires a few processing steps before it can be used in the Indirect AR Browser.

4.1 Integration of OpenStreetMap Data

We store the 2D map data exported from OpenStreetMap in a Postgres database. We can query the database from the browser using HTTP requests to the serverside Node.js application described in section 3.2. Using the user's latitude and longitude as parameters, we make a request to server. The server then returns the details of the all the buildings and point features within a 100 meter radius of the user.

Since the building outlines are stored as 2D vectors in the database we can easily recreate them as 2D polygons in Three.js. As the OSM data is stored using the WGS84 coordinate system we must then convert them into a local coordinate system before we can place these polygons in our scene. We can then take these 2D building outlines and extrude them upwards to create sparse 3D models of the buildings. These models can be used to position our labels in the scene once we align them with the buildings in the image.

4.2 Alignment of 3D Content

One of the key advantages of indirect augmented reality is that we can align our real and virtual content without relying on computer vision techniques. Instead we rely on the information we already have about our real and virtual environments, creating a mapping between the two. Each of the images we retrieve from Mapillary includes the latitude and longitude reported by the camera, along with an angle measuring the cameras rotation from north. OSM provides the coordinates for each object retrieved. We place our virtual camera at the origin point of our scene and then treat the origin as having the latitude and longitude of the retrieved image. This allows us to position the buildings in the virtual scene based on their location relative to the real camera.

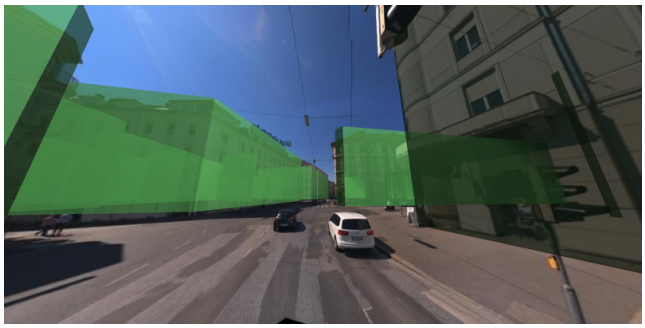

Figure 3: Correct alignment between the 3D content and the indirect AR scene.

The next step is ensuring the correct rotation around the vertical axis to properly align our real and virtual buildings. We use the camera angle found in the images metadata to apply a rotation around the vertical axis and align north in the image with north in our map data (Figure 3). Provided the metadata is sufficiently accurate this should provide a matching close enough to not produce any recognisable alignment error.

There are three types of error encountered in the current implementation of this alignment method. The first occurs due to inaccuracy in the camera angle measurement. This results in an incorrect rotation around the y-axis. In one of the image sets we used for testing, the images were captured using a camera mounted to a moving vehicle, and this type of error was found in images taken as the vehicle carrying the camera turned a corner. This type of error is entirely dependent on the quality of the data provided by Mapillary. It is worth noting Mapillary allows users to manually change the camera angle for any photo.

The second type of error occurs when the camera was not level as it captured the image, resulting in the scene appearing to be rotated by some amount of pitch and roll. Buildings in the image appearing to be slanted and lines that should be perfectly vertical or horizontal appear angled. We cannot account for this error creating a misalignment with our virtual content. Mapillary does not provide functionality to let users adjust the rotation of an image around

[4]https://www.postgresql.org/

[5]http://www.qgis.org

[6]http://postgis.net

these axes, so this type of error is commonly found with images captured using a handheld camera. Figure 4 shows an alignment error caused by a combination of camera tilt and error in the camera angle measurement.

The final type of alignment error is due to the fact we do not account for elevation differences in the environment. All our buildings are assumed to be sitting on a flat plane meaning all our buildings are at the same height, resulting in our virtual buildings appearing above or below the real buildings in scenes with significant differences in elevation between buildings. The incorporation of elevation data from another data source could be a possible solution in future work.

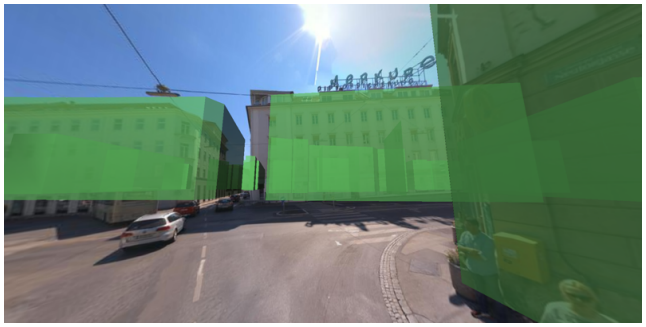

Figure 4: Alignment error due to camera tilt and inaccurate camera angle.

5 Visualisation

After accessing and aligning the data, the next step is the visualisation of the GIS data in the Indirect AR Browser. For this purpose, we investigated different ways of how to visualise the different data.

5.1 Label Visualisation

We place our labels dynamically in the scene to ensure maximum visibility and legibility. We make use of the coarse 3D building geometry that we have created from the building outlines provided by the GIS data. The coarse 3D building geometry in combination with the user's position allows us to place labels that are clearly within the view of our virtual camera. The main idea behind our alignment approach is that for each building in the user's view, we want to find the face that is most visible to the user. In addition we position a 3D text label parallel to this selected building face to give the effect that the label is anchored to the wall of the building as shown in Figure 1, Left and Middle. This effect helps to make up for the absence of depth queues in our scene, helps users understand how the label is positioned spatially in relation to the real environment [13], and provides a clear visual relationship between the building and the annotation [8].

We devised a method of selecting the best candidate face of each building in the scene. First we want to check whether the face is occluded by another object in the scene. We make use of a raycasting operation to check whether a face is obscured by another building. This involves casting a ray from our camera to the centre of the face, and checking if the ray is intercepted by any other objects in the scene. Any face that is occluded is not a suitable candidate for label placement.

We also consider the angle of the face to the camera. If the label is aligned with a face that is at too sharp an angle to the camera the text will become illegible as shown in Figure 5. We can measure the angle between the face and the ray we cast previously. The closer this angle is to 90 degrees the more legible the text will be. At 90 degrees the text is completely flat to the camera, providing maximum legibility. By using a set threshold of approx. 60 degree, we avoid that building faces with a low legibility will be selected.

Figure 5: The right label is less legible due to the angle.

The width of the face is also taken in to consideration. A face that is too narrow will not provide adequate room to position a label in convincing fashion. Faces that are narrower than a defined threshold of 2 meters are simply ignored and not considered to be candidate faces.

While we do make use of the *building:levels* tag in the OSM data to estimate the height of each building we simply place labels at eye level. Future iterations of the application could involve adjusting label heights the user pans their view up and down, but the majority of the time the user will be viewing the scene at eye level.

5.2 Visualisation of Point Features

Point features in OSM are objects of interest that are stored as a single node, with a single latitude/longitude pair. Point features include a vast variety of objects, including bus stops, restrooms and landmarks such as statues. In our test area the density of point features is so high that attempting to display all of the point features within a users line of sight would result in a completely illegible mass of annotations. This problem is easily solved by simply allowing users to filter which type of point features they want visualised and adjust the database queries based on these filters, this is similar to the knowledge based filter proposed by Feiner et al. [2] where the information displayed is based on the users current task. For our application we have simply limited the features we have exported from OSM, as we just want to investigate how they can be represented in a 3D environment. For testing we chose to create markers for bus stops, as they are suitably common in our test area and are positioned a reasonable distance from each other.

We first decided to mark point features using a floating sphere with an annotation overhead. Since these annotations would always be facing towards the viewer we had no need to use 3D text, as we did with our building labels. We chose to use to simple 2D billboard style annotations for our point features. Multiple studies into nability in 3D environments [3, 5] found this was the most commonly used way to integrate text over a 3D scene to provide optimal readability in conditions negatively affected by background texture and illumination.

Once we integrated point features into our scene we found the lack of any depth queues made it impossible to discern where the marker was positioned in 3D space. We can discern depth with our building labels as they are positioned to look like they are attached to real objects in the image [13]. We provide a similar effect with our point features by finding a way to connect them to our ground plane. We achieved this by drawing simple shadows directly below our markers, at the level of our virtual ground plane. This provides the extra depth information needed for us to identify where the marker sits relative to the objects in our image. These shadows are not always ideal. They can be difficult to spot against dark or noisy backgrounds, and the user will have some expectation about

how shadows should look or be positioned based on the lighting conditions in the image. In Figure 1, Right the shadow below the left label blends into the real shadow in the image, making it harder to discern depth.

6 Performance Analysis

One of the biggest concerns for the Indirect AR Browser was the time taken to load a new image and create the corresponding 3D content (such as 3D buildings and annotations), especially because we were targeting less powerful mobile devices. Our performance is tied to how we construct 3D content, but also to the backing infrastructure we provide and the APIs we rely on. For testing we used an Apple iMac with an Intel Core 2 Duo clocked at 3.1GHz and 8GB of RAM and an Apple iPad 3 with a 1GHz dual-core ARM CPU and 1GB of RAM for the mobile experience.

6.1 Data Retrieval

In the Indirect AR Browser when moving to a new location, we need to retrieve data from three different sources. First we have to query the Mapillary API to retrieve the image feature closest to our location. Second we need to load that image from the Mapillary server. Finally we query our database to retrieve GIS data for the area surrounding our position.

The time taken to retrieve the image features through Mapillary's API remained fairly constant, with a average time of 680 milliseconds when testing on a desktop PC. When testing on an iPad we found the average time taken increased to 1633 milliseconds. It is worth noting that the desktop PC was using a wired Internet connection while the iPad was connected over a local wireless network.

The images provided by Mapillary are stored on an Amazon Cloudfront server. We use the image key provided as part of the image feature to construct a Cloudfront URL and retrieve our image. We passed the image URL to the Three.js TextureLoader and measured the time taken for that function to complete. We recorded an average time taken of 4300ms on our desktop PC and an average of 3900ms when using the iPad. These times were highly variable, with the load times on the desktop PC ranged from 3003 milliseconds to a maximum of 6302 milliseconds.

The PostGIS database extension allow us to query for all the buildings and point features within a set radius of our new location. The number of buildings we retrieve affects the time taken to reconstruct our scene. After testing a variety of different settings we settled on a 100 meter radius as a balance between performance and retrieving enough buildings to adequately annotate a scene. Past 100m meters the extra buildings are too distant to be effectively labelled or are almost entirely occluded by closer buildings.

6.2 Content Construction

Once we have retrieved a set of buildings from the database we convert their coordinates into a local coordinate system so we can position them in relation to our user's position and create 3D geometry. We then apply our discussed method for selecting the best candidate face for labelling by iterating over each face in the scene. To analyse the performance of our scene construction we measure the time taken to create the scene and graph it against the number of faces or the number of buildings in our scene. The number of faces provides us with a measure of geometric complexity for the scene, while the number of buildings lets us know how many objects need to be retrieved from the database and converted into geometry.

When using a fixed retrieval radius of 100 meters and retrieving images at random from our test set we found the variation in building density had no effect on the time taken. This is due to the fact most of the faces fail to pass the first check of our selection function and are quickly ignored as possible candidates. We found a noticeable difference when we increased the retrieval radius from 100m to 300m, with the scene construction taking on average 450 milliseconds longer. However any increase beyond 100 meters does not provide any benefit to the user, as buildings at this distance are too far from the user to be labelled clearly. Figure 6 shows the average time for each load over 30 trials, with the average total load time being 6361 milliseconds.

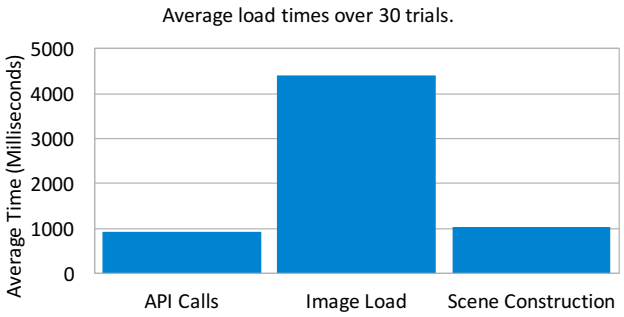

Figure 6: Average load times as we select random areas from our test set.

6.3 Runtime Performance

To test the performance to our application we set the camera to constantly rotate, forcing the renderer to update constantly with a maximum frame rate of 60 frames per second. We tested performance across multiple browsers and on both a desktop PC and a 3rd generation iPad. Table 1 shows the average frame rate across the browsers using the same scene with 5 labelled buildings.

Browser	Resolution	Average frame rate
Firefox	1920x971	30 FPS
Firefox	1920x1080	30 FPS
Chrome	1920x974	45 FPS
Chrome	1920x1080	60 FPS
Safari	1920x1019	60 FPS
Safari (iPad)	980x1225	60 FPS

Table 1: Performance measurements.

Performance in Firefox was the poorest of the browsers we tested, with a constant 30 FPS in both windowed and fullscreen modes. Chrome provided a constant 60 FPS in fullscreen mode, but when displaying the application in a window the frame rate was highly variable, with an average of 45 FPS. The variance in frame rate resulted in panning around the scene appearing choppy. Safari offered the best performance overall, providing a constant 60 FPS across both mobile and desktop versions, fullscreen and windowed.

6.4 Discussion

For our application to be usable in real world conditions the time taken to retrieve and new image and construct the new scene needs to be low enough to provide a sufficient update rate as the user moves around the world. If the application takes too long to update as the user changes position the application can seem unresponsive and the issue of spatial inconsistency arises, leading to a poor user experience. This time requirement is tied to how often we choose to update the scene. We request the user's GPS coordinates at fixed intervals and check the distance they've moved from the centre of the last 3D scene we constructed. If this distance is greater than some set threshold we request that the scene is updated with a new image.

While the time taken to construct the 3D scene and the time taken to retrieve an image feature from Mapillary are within acceptable

bounds, the time taken to load an image takes the majority of time, as seen in Figure 6. While we have not conducted any form of user testing so far, we believe the improvements in load times will positively affect the user experience.

The application runs at an acceptable frame rate on all the devices we tested, although Safari was the only browser to provide a full 60 FPS. In Firefox and Chrome (when not in fullscreen) the rotation of the camera did appear choppier than the frame rate would suggest, likely due to inconsistencies in frame timing within the space of a second. Considering the low power of the devices we tested on we consider the performance of our application to be acceptable.

7 Conclusions and Future Work

This work proposes an indirect augmented reality approach to address several issues with situated visualisation applications, including the limited understanding the real world environment and issues with visibility and legibility of data. Our application functions well under what we consider to be ideal environmental conditions. This includes using images that are taken under ideal conditions, having accurate metadata about the image, and having accurate GIS data for the area.

Our alignment method relies heavily on the accuracy of the reported compass angle and latitude/longitude. Since the used provider for street level imagery (Mapillary) offers no way for users to manually adjust the rotation of the image, except around the vertical axis, correction for the pitch and roll of the camera can only be done within our application. Future work could involve using an image based approach to adjust for this rotation. This would likely involve using edge detection techniques to produce an edge map where the outlines of the pictured buildings are clearly defined, we could then find the rotation that best matches the edges of our virtual buildings with those in the image.

We are reliant on the quality of the GIS data provided by OpenStreetMap. When a building lacks a name tag in its metadata we default to using the buildings address as the label, but this information is also missing from some buildings in our test area. To produce a fully annotated scene we require the buildings in the area to be adequately tagged in OSM.

The final requirement for an ideal environment is the lack of elevation changes in the scene. As discussed our approach assumes all the buildings in the scene are sitting on a flat plane. This means that if we use an image where the buildings sit on an incline our virtual content will not match the elevation of the real buildings. Future work could include investigating the use of a digital elevation model to adjust the elevation of each building in our scene, however it is unclear whether the availability and accuracy of this data is suitable for our purpose.

While our application does address the issues we outlined, further work could be done to produce a more robust application that functions ideally under a wider range of conditions.

Our label alignment work could be further improved by combining our use of GIS data with an image-based approach. The image based techniques discussed by Grasset et al. [4] can be applied over our 360 degree image to find areas of the image that are suitable for labelling. Significant work would be required to combine both of these approaches, as one operates exclusively on the image while the other operates exclusively on our 3D scene.

Further work can also be done to dynamically adjust the position of labels as the user pans around our scene. Our program positions each label when a new image is loaded and the labels remain static as the user pans around the image. If we reposition labels dynamically we can deal with labels that are partially out of frame, or labels that are out of frame entirely while their associated building is still in view of the camera.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1464420.

References

[1] B. Bell, S. Feiner, and T. Höllerer. View management for virtual and augmented reality. In *Proceedings of the 14th annual ACM symposium on User interface software and technology - UIST '01*, 2001. doi: 10.1145/502348.502363

[2] S. Feiner, B. Macintyre, and D. Seligmann. Knowledge-based augmented reality. *Communications of the ACM*, 1993. doi: 10.1145/159544.159587

[3] J. Gabbard, J. Swan, D. Hix, R. Schulman, J. Lucas, and D. Gupta. An empirical user-based study of text drawing styles and outdoor background textures for augmented reality. In *IEEE Proceedings. VR 2005. Virtual Reality, 2005.*, pp. 11–317. IEEE. doi: 10.1109/VR.2005.1492748

[4] R. Grasset, T. Langlotz, D. Kalkofen, M. Tatzgern, and D. Schmalstieg. Image-driven view management for augmented reality browsers. In *ISMAR 2012 - 11th IEEE International Symposium on Mixed and Augmented Reality 2012, Science and Technology Papers*, 2012. doi: 10.1109/ISMAR.2012.6402555

[5] J. Jankowski and K. Samp. Integrating Text with Video and 3D Graphics: The Effects of Text Drawing Styles on Text Readability. In *Proceedings of the 28th international conference on Human factors in computing systems*, 2010. doi: 10.1145/1753326.1753524

[6] S. Kurkovsky, R. Koshy, V. Novak, and P. Szul. Current issues in handheld augmented reality. In *International Conference on Communications and Information Technology - Proceedings*, 2012. doi: 10.1109/ICCITechnol.2012.6285844

[7] T. Langlotz, T. Nguyen, D. Schmalstieg, and R. Grasset. Next-generation augmented reality browsers: Rich, seamless, and adaptive. *Proceedings of the IEEE*, 2014. doi: 10.1109/JPROC.2013.2294255

[8] S. Maass and J. Dollner. Dynamic Annotation of Interactive Environments using Object-Integrated Billboards. In *14th International Conference in Central Europe on Computer Graphics Visualization and Computer Vision WSCG*, 2006.

[9] F. Okura, T. Akaguma, T. Sato, and N. Yokoya. Addressing temporal inconsistency in indirect augmented reality, 2016. doi: 10.1007/s11042-015-3222-0

[10] F. Shibata, H. Nakamoto, R. Sasaki, A. Kimura, and H. Tamura. A View Management Method for Mobile Mixed Reality Systems. *EGVE'08 Proceedings of the 14th Eurographics conference on Virtual Environments*, 2008. doi: 10.2312/EGVE/EGVE08/017-024

[11] S. White and S. Feiner. SiteLens: Situated Visualization Techniques for Urban Site Visits. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pp. 1117–1120. ACM, New York, NY, USA, 2009. doi: 10.1145/1518701.1518871

[12] J. Wither, Y.-T. Tsai, and R. Azuma. Indirect augmented reality. *Computers & Graphics*, 35(4):810–822, 2011. doi: 10.1016/j.cag.2011.04.010

[13] S. Zollmann, C. Hoppe, T. Langlotz, and G. Reitmayr. FlyAR: Augmented Reality Supported Micro Aerial Vehicle Navigation. *IEEE Transactions on Visualization and Computer Graphics*, 2014.

[14] S. Zollmann, C. Poglitsch, and J. Ventura. VISGIS: Dynamic situated visualization for geographic information systems. In *International Conference Image and Vision Computing New Zealand*, 2016. doi: 10.1109/IVCNZ.2016.7804440