DreamStore: A Data Platform for Enabling Shared Augmented Reality

Meraj Khan* Arnab Nandi†

The Ohio State University

Abstract

Unlike traditional object stores, Augmented Reality (AR) query workloads possess several unique characteristics, such as spatial and visual information. Such workloads are often keyed on a variety of attributes simultaneously, such as device orientation and position, the scene in view, and spatial anchors. The natural mode of user-interaction in these devices triggers queries implicitly based on the field in the user's view at any instant, generating data queries in excess of the device frame rate. Ensuring a smooth user experience in such a scenario requires a systemic solution exploiting the unique characteristics of the AR workloads. For exploration in such contexts, we are presented with a view-maintenance or cache-prefetching problem; how do we download the smallest subset from the server to the mixed reality device such that latency and device space constraints are met?

We present a novel data platform — DreamStore, that considers AR queries as first-class queries, and *view-maintenance* and large-scale analytics infrastructure around this design choice. Through performance experiments on large-scale and query-intensive AR workloads on DreamStore, we show the advantages and the capabilities of our proposed platform.

1 Introduction

Over the past few years, the field of Augmented Reality has transitioned from a science-fiction concept to a consumer-grade technology used by millions [1, 2]. AR has found a variety of specific use cases including gaming [3], architecture [35], education [36], medicine [8], and emergency services [11, 34], each enabling an entirely new way of digital interaction using camera-based devices. This progress is driven by the advancement in sensing, computing, and communications technology. Despite these developments, the reach and the state of AR applications today are not up to its potential [26]. There are only a few platforms [19, 22] with tools for building shared reality experiences where multiple users with different devices can share the same augmented experience.

The significant obstacles hindering a wide-scale adoption of AR are its unique user-interaction characteristics, data management needs [29], and platform fragmentation [18]. Augmented Reality exacerbates some of the challenges posed by traditional user interfaces [29]. For example, AR applications can generate queries at a rate much higher than traditional user interfaces, often higher than the frame rate of the device camera, and the workload is bursty. Users interact with the scene-in-view through the means of implicit and explicit gestures (Section 4). In AR scenarios, the image capture is constantly changing (limited by device frame-rate and capture-rate): each image reflects the set of "queries" posed by the user. Each image in the field of view of the user can potentially contain several points of interest for which AR overlay information is available in the datastore.

*e-mail: khan.485@osu.edu

†e-mail:arnab@cse.osu.edu

In a shared reality setting, the workload generated at such a high rate by multiple clients can overwhelm the backend, and result in dropped queries and lags on the user interface. Figure 4 shows the query rate every 100 ms while using our prototype room scanning app. There is considerable variability in the query rate throughout the session, going from 0 queries in some intervals to up-to 13, which corresponds to a query rate of $130/s$ — more than twice the frame-rate for most devices. Another major concern is overwhelming the user with too much information, which is more aggravating on AR interfaces [29]. At any instant, a large number of objects in the field of view can be potentially augmented, which could lead to visual clutter [25]. Visual clutter on AR interfaces does not just degrade the user experience but is capable of causing physical harm and material damage to the users and their surroundings.

Contributions: We propose DreamStore — an AR application data platform for addressing these issues. DreamStore provides a data-management API for application developers to model AR application workflows facilitating porting of core functionalities between different client platforms. We provide effective workload reduction and prefetching strategies for AR applications that facilitate interactive latency and do not overwhelm the UI with insignificant details. We provide two query-intensive workloads that emulate real-world AR application usage and present their performance evaluation on the DreamStore platform.

The platform incorporates optimizations for AR workload characteristics at various layers of the data stack. The query workload generated by the user interface is reduced by inferring query priorities from user gestures and dropping low priority queries. This makes sure that the users' field-of-view is not cluttered as the system ignores low-interest objects, and does not waste resources fetching and rendering unimportant object details. The clients maintain a local cache with information about AR objects in their vicinity to reduce network roundtrips. This cache is maintained without explicit intervention from the developer through a pub-sub mechanism for syncing updates on object information across clients and prefetching information about objects likely to be queried in the near-future based on the device's position, orientation, and trajectory. While there are existing works that support data infrastructure for games and virtual reality [12, 23, 33], the inclusion of rapidly changing query *likelihoods* derived from computer vision and the query session, alongside a multi-user setting, confounds the infrastructure problem. There is a need for a scalable infrastructure solution to provide a shared experience where users with different AR devices can interact and manipulate AR objects in a localized environment.

Motivation: DreamStore can power shared AR experiences in a variety of settings with multiple users accessing and interacting with the same AR objects through their personal AR client devices. It can enhance the functionality and capabilities of the emerging use-cases in shop floors [9], warehouses, and Computer Supported Collaborative Work (CSCW) systems with AR interfaces [28]. For example, dedicated tablets fixed around entrances to conference rooms, classrooms, etc for displaying the schedule have become common in workspaces and schools. These devices support different level of interactivity ranging from allowing users to just see the current schedule to letting them update it. Although convenient, this solution requires significant investment in terms of dedicated hardware and installation effort. A potential solution to this could be enabling users to access this information on their personal devices through an augmented reality application. Although this seems like a trivial scenario, the data management problem intensifies in presence of a large number of users simultaneously accessing information on common rooms. A significant delay in the propagation of information update by a user to the other users simultaneously accessing it, and lag in information retrieval by an overwhelmed system can degrade the user experience. A similar use-case imagined in a different setting such as a mall during special events can enable the stores to convey geo-tagged promotions, advertisements, updates, etc. in realtime to the shoppers. It can also enable building collaborative/competitive multiplayer AR games [4, 13] in a shared physical setting by providing an efficient data storage and synchronization framework for player-AR object interactions.

There have been several impressive improvements in computer vision (CV) research [31] recently, to the point where reasonably advanced techniques are now available as reliable building blocks for other research. Furthermore, augmented reality (AR) wearable devices such as Google Glass and Microsoft HoloLens, have become available, which continually capture and process image and video data and provide pertinent feedback (i.e., the augmentation) through an overlay display. These devices have inspired and unlocked a variety of "camera-first" interaction modalities, where the camera is often the primary mode of capture and input. This paradigm is transferring over to smartphones and tablets as well. AR applications such as Snapchat, Google Lens, and Amazon Shopping are bringing a completely new and natural mode of interaction to consumer-grade smartphones and tablets [15].

Smartphones and tablet devices have become extremely affordable at the mass-consumer scale and capable of either on-device or on-cloud image processing. When backed with large data stores, they can serve as the ideal edge device, for uses as broad as education, workforce training, healthcare, enterprise, and manufacturing. Thus, there will be a critical need for data infrastructure support to meet this trend to serve the workloads generated by these devices. The number of end-user activities that are backed by large amounts of data is rapidly increasing. For example, a simple restaurant look-up on Google Search and Google Maps is augmented with wait times, popular hours, and visit duration, aggregated from population-scale user location history logs [6]. The data-rich paradigm — considering a user's AR view as a queried view on a large data warehouse brings about a unique opportunity to build compelling experiences for end-users.

Given the computational capabilities of the end-device and the network limitations, a critical consideration for AR applications is roundtrip latency. Since our AR device is a commodity phone or tablet, we expect the system to acquire the camera feed, preprocess it, extract structured information, query it against a database over the network, and return it as an image overlay — all in real-time. Clearly, given standard frame rates, a smooth end-user experience will necessitate algorithmic contributions that minimize network roundtrips and take advantage of unique properties of the AR workloads.

With the advent of Augmented Reality technology on common mobile devices, we have a wide variety of AR-capable devices running on different platforms, each supporting a different set of functionalities. Although we have development environments like Unity and libraries like Vuforia, ARToolkit which support development for all popular platforms — Android, iOS, UWP (including Hololens), these applications are not easy to port from one platform to the other. Moreover, there is no infrastructure solution to provide a shared experience where users with different AR devices can interact and manipulate AR objects in a localized environment.

2 Related Work

Data Platforms for Augmented Reality workloads: Schmaleteig et al. [27] present a 3-tier data model for handling AR workloads. This work demonstrates the usage of context-sensitive scene graph data to construct views for AR apps from large databases of GIS XML data. The database layer is linked to the application layer by a data transformation layer, which maps raw data from the database to specific object types. Similar to DreamStore, it decouples the presentation layer from the data layer. The middle-tier is similar to the client-specific ARView management library in our system.

Nicklas and Mitschang [21] propose a 3-layer model, including the client device layer, federation layer, and server layer. The server layer stores resources for the entire system, including geographical data, users' locations, and virtual objects. The federation layer, similar to the system developed by Schmalteig et al., provides transparent data access to the upper layer using a register mechanism by decomposing the queries from the client layer and then dispatches them to registers for information access.

LightDB [14] is a multidimensional array database for virtual reality applications utilizing orientation prediction to reduce data transfer by degrading out-of-view portions of the video. In contrast, the DreamStore system is built for AR applications exposing only the AR object information as opposed to storing and processing an entire scene.

Microsoft's cloud service — Azure Spatial Anchors [19], provides a shared persistent database for mixed reality objects that can be accessed by multiple client platforms. The service stores the AR objects created by users keyed on their sparse spatial scans, unlike DreamStore, and relies on the cloud infrastructure to service all `READ` and `WRITE` requests without a managed local cache, similar to the baseline setup in our evaluation. Since the only possible way of querying on the Spatial Anchors platform is through spatial scans, a direct comparison with DreamStore is not possible as our querying API is based on pre-recognized objects. Moreover, without the pub-sub paradigm, the clients do not actively listen for updates on the objects they are currently accessing.

Shared AR and Interactivity Singh et al. [29] present a study of Augmented Reality systems in which they describe a generic architecture for an AR application. This architecture does not consider the problem of synchronization in a multi-client environment. Slay et al. [30] describe querying for virtual objects in an AR application using fiducial markers and other user interactions. They do not address the scaling problem in a rich localized-environment. Recent works in collaborative AR [28] and shared AR world game design [4] are focused on placement and tracking of AR objects in the shared world and interaction and experience design around, which are orthogonal to our work.

Location-based Prefetching for Mobile Applications: Geiger et al. [10] introduced ideas for developing a generic location-based mobile augmented reality. Their focus is on adding AR objects and tracking them through a user session based on camera-feed and device position employing smart prefetching algorithms based on motion profiles. This work is more in line with the now freely available AR toolkits that we use in DreamStore. This is in contrast to our effort of scaling and enabling data synchronization in a shared augmented reality setting. `IMP` [16] aims to shift the prefetching burden from applications to mobile-systems in order to optimize for energy and cellular data usage. DreamStore is designed around managing data needs for AR applications powered by application-specific databases, properties of which the mobile systems cannot exploit. However, an `IMP` like system could potentially enhance the performance of DreamStore applications by introducing an additional prefetching layer if it can speculate the applications' needs.

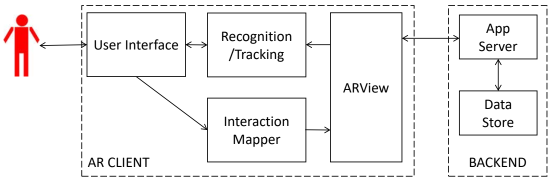


Figure 1: DreamStore modules at the Augmented Reality client devices and the backend. The figure shows the interactions between different modules.

3 System Overview

DreamStore is a data platform for AR applications. Figure 1 depicts the modules of the system and the interaction between them.

The backend data store consists of a key-value store that houses the information on the AR objects keyed on their identifier. The recognition/tracking module identifies points of overlay in the scene for AR objects. The application developers can choose from the various toolkits available in the market, e.g., ARToolkit, ARCore, and Vuforia, for this purpose.

Users can interact with the AR system through either *implicit* gestures such as inadvertently pointing their AR device in a certain direction at a certain location without prior knowledge of the available AR objects, or they can employ *explicit* gestures such as tap-and-interact or a voice command to interact with a known AR object. The user interactions through implicit and explicit gestures trigger object detection and tracking module, which identifies spots in the view for which there is information available in the local or remote database. The tracking module further maintains the positions for the different overlays and tracks them throughout the session for correct placement of the retrieved data. Libraries like ARToolkit and Vuforia make use of a unique set of marks or images called *fiducial markers* to identify and track these points in the physical space. The image/feature database for these markers can be stored locally or in an online database. For the sake of simplicity, we depict the recognition/tracking module housed entirely on the client-side in Figure 1. Note that this feature/image database for fiducial markers is different from the backend datastore, which maintains only the AR information to be overlaid.

The user gestures combined with the object identification information generated by the recognition/tracking module are processed by the Interaction Mapper which maps a seemingly large number of queries to a well-defined reduced set of API calls (Section 4.1) to the client-side cache- ARView. Additionally, the interaction mapper module uses the client device's current position, orientation, and direction of motion to generate the set of object identifiers that are likely to be queried next and fetches them from the server to be stored in the ARView.

ARView is maintained using a proximity-based cache replacement policy. When the cache is at capacity, the objects farthest from the most recently queried object are replaced first. These API calls may trigger data fetches or updates from/to the backend server. The communication between the client-side cache — ARView and the backend store is managed by the view-management modules at the client-end and the server-end without any intervention from the application developer. The ARView is maintained asynchronously with updates from the backend server in real-time.

4 Querying in AR

DreamStore handles virtual object querying in an AR environment through the means of implicit and explicit user gestures, or device position and orientation. All queries are mapped to DreamStore API (Section 4.1) calls, which do not distinguish between the query modality and only specify the level of details requested for any specific object.

We elucidate the API call mapping and queried object prioritization through two basic gestures commonly covered by most visual AR interfaces – *gaze* and *tap*. The implicit *gaze* gesture triggers a read on the set of AR objects in the field of view. Additionally, it generates the client device's current position in physical space, orientation, and direction of motion. A detailed discussion of indoor localization techniques for generating the device position is beyond the scope of this work.

AR applications can potentially generate data queries at a rate higher than the device frame-rate (one query per AR object in the field of view). This can overwhelm the network and the backend data warehouse. We use ideas from existing work on interactive querying [7] to reduce the query workload. The user's query intent is mapped to object priorities. The UI generates a '`READ`' request for every object in the field of view. The interaction mapper periodically aggregates these requests and assigns them priorities according to the inferred user interest. High-interest objects generate more `READ` calls every period. The system drops the requests on objects with `READ` below a certain developer-defined threshold on the priority.

4.1 DreamStore Data Interface

The AR applications running on the DreamStore platform interact with the local client-side cache ARView through data management API calls. These API calls can trigger communication with the backend data stores. The application developer is oblivious to this communication, and it is managed by the view-management modules on the client and the backend. The API supports read-write operations on the AR object properties. The cache is kept updated by subscribing for updates on every object queried by the client maintained in its ARView.

- `READ` ($\{object\_ids\}$)– This augments the queried objects with the information from the ARView, and generates subscription requests for updates on these objects.
- `WRITE` ($object\_id : object\_value$)– The editable fields (information from the primary key-value store) in the AR object - $object\_id$ are updated with the value $object\_value$. The ARView stores a '*dirty*' bit for each object and sets it when the `COMMIT` function is called on an object.
- `COMMIT()`– All the ARView objects with a set dirty bit are pushed to the backend key-value store.

4.2 Query Priority Mapping

We build upon the prior work on interactive querying [7] and propose an AR-specific query-reduction technique: it models the user's query intent as an expectation over the field of likely queries. The user interface generates a `READ` query for each of the identified objects in the frame and passes it onto the Interaction Mapper module. The interaction mapper periodically aggregates all these `READ` requests and calculates the density or the count for each object query. The application developer has the option of configuring either density or count as the metric of object importance.

A user is likely to focus on an object of greater interest for more time than the other objects, consequently generating more `READ` requests on it. The mapper calculates the density for each object O_i from the set of objects – $\{O_1, O_2, ..., O_n\}$ queried as $\rho_i = \frac{|O_i|}{\sum_{i=1}^{n} O_i}$.

The number of queries on an object O_i in a batch is given by $|O_i|$. Explicit query on a specific object O_i by the user, potentially triggered by object selection using a tap gesture or other means depending on the interface, sets the query density for that object as $\rho_i = 1$ and abandons queries on the other objects. This is because an explicit selection of an object by the user implies the user's primary interest in the object.

Using object density as the metric of object importance has the potential of overwhelming the system in some cases. If a user scans through the space around them in a manner such that at any time, there are only a few AR objects in the view and the objects in the field of view change at a rate near to that of the interaction mapper's aggregation rate, all the objects will get a high density score which would translate to high importance. Also, if the physical space has a lot of AR Objects, and the user continues expressing interest in them by keeping them in view, the system would still assign lower importance to the objects because of their uniformly lower densities despite being queried frequently. We provide *count* as another metric of object-importance which can handle such situations. The interaction mapper interprets the query-intent as a discrete numeric function, where each point corresponds to the number of times the recognition module issues a `READ` call on an object. The argument for higher counts corresponding to greater user interest is the same that we made for density. The user intent function – $\boldsymbol{I}$ is mapped to either the density function ρ or the absolute object counts $\mathbf{c}$.

The interaction mapper first performs thresholding on the intent function generated by user interaction $\boldsymbol{I}$. All objects with an intent value lower than a developer configured threshold – I_{MIN} are discarded. The mapper then translates the set of intent values $\boldsymbol{I}$ to a set of *READ* ARView API calls. The mapper translates the intent function into up to three sets of `READ` calls with high, medium, and low priorities. The mechanism used for this mapping is developer-configurable, and the developer can choose from either *value thresholding* or *proportion thresholding*.

For value thresholding, the user defines a minimum intent threshold for high-priority assignment – I_{HIGH} and a minimum intent threshold for medium priority assignment – I_{MEDIUM}. The objects with intent values higher than I_{HIGH} are mapped to a `READ` call with `HIGH` priority. Of the objects with no assigned priority, the ones with intent values higher than I_{MEDIUM} are mapped to a `READ` call with `MEDIUM` priority. The remaining objects are mapped to `LOW` priority `READ` calls. In this method, the developer defines the proportions of the thresholded set to be mapped into `HIGH`, `MEDIUM`, and `LOW` priority `READ` calls.

The thresholding techniques presented here are not optimized for a large list of objects. Standard *top-k* techniques for optimizing look-up time can be used for environments that are immensely rich in augmentation. Similarly, other modalities of querying, such as *voice* can be mapped to DreamStore API calls by applying modality-specific mapping and reduction logic.

5 PREFETCHING FOR INTERACTIVITY

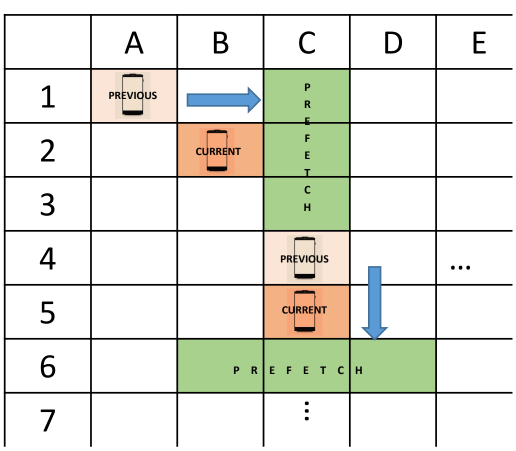


Figure 2: Directional Prefetching Strategy based on the four principal directions of movement. The green regions indicate the regions to be prefetched based on the current and the previous location.

The interaction mapper predicts the set of AR objects to be prefetched from the backend data store to keep the DreamStore client interactive, by ensuring objects that are likely to be queried in the near-future are available in the client-side cache.

We describe the prefetching mechanism for a generic Indoor Positioning System (IPS) solution. This abstract mechanism can be customized for different localization techniques depending on their capabilities. All AR objects are assigned to a unique bounded physical space identified by a *region_id* in the AR setting. For example, each cubicle in a large office, each room ,or each floor in a building can have a unique region_id. This is a design choice and would depend on application requirements such as the density of AR objects in the environment, and typical movement pattern and coverage of users.

At the end of every thresholding period, the interaction mapper performs a `PREFETCH` operation. The prefetching logic generates a set of *object_id*s, and all these objects are queried at a low priority. The specifics of the `PREFETCH` operation would depend on the IPS capabilities.

Function `prefetch`(*position_context, motion_context*):
  prefetch_region $\longleftarrow$
   generate(position_context,motion_context)
   prefetch_set $\longleftarrow$ *region_map.get*(key = prefetch_region)
  return *prefetch_set*

Algorithm 1: Generic prefetching logic to identify the AR objects to be prefetched based on the position and the movement of the client device

Algorithm 1 describes a generic prefetching logic utilizing the client device's *position_context* and *motion_context*. The *generate* logic is entirely dependent on the deployed IPS solution. For example, a low-effort solution can map AR objects to specific *region_id*s, and for each `PREFETCH` operation, it can return the list of objects which are present in the same region. In this case, the *generate* logic does not utilize the *motion_context* as it is not available. This solution would work well when the users are confined to the same region for the most part. However, use-cases with movement-patterns spanning multiple regions or frequent switching between regions would generate a large number of `PREFETCH` requests, which are barely used and hamper the system performance.

The prefetching mechanism can be improved by movement prediction. A working movement prediction logic can be easily implemented by banking on spatial locality in the access pattern of AR applications. A directional prefetching logic can *generate* region identifiers closest to the current *physical_context* of the device in the direction of its current motion. The direction of motion can be identified by various techniques such as extrapolating trajectory from recently logged location data, using on-device motion sensors, or a fusion of these and other techniques. The `PREFETCH` operation is performed after the regular read batch is generated by the interaction mapper. This ensures that the prefetching requests are queued after the read requests for objects that have been explicitly queried.

Some IPS solutions can effectively map large physical spaces onto a grid with distinct regions or cells. DreamStore uses one such localization dataset to emulate an AR workload. A simple directional prefetching strategy in such a scenario can be implemented by tracking just the device's current location (cell in the grid) and the last recorded location, using them to determine the principal direction of motion (*up, down, left,* or *right*). The current cell provides the *position_context*, and the last recorded cell, along with the current cell, provides the *motion_context*. We track the change in the cell position to predict the next likely regions the device will move to. Figure 2 shows this prefetching strategy in action. The device currently in cell *B-2* was located previously in *A-1*, and hence the principal direction of motion is determined to be *right*, and the *generate* (Algorithm 1) logic would return the regions adjacent to the current location towards the right — *C-1*, *C-2*, and *C-3*.

IPS technologies regularly use crowdsourcing for improving indoor positioning accuracy and mapping sensor profiles to physical spaces. These profiles can be used to train predictors for determining the next spatial region the user might move to, given the movement trajectory until that point.

6 AR-View Maintenance

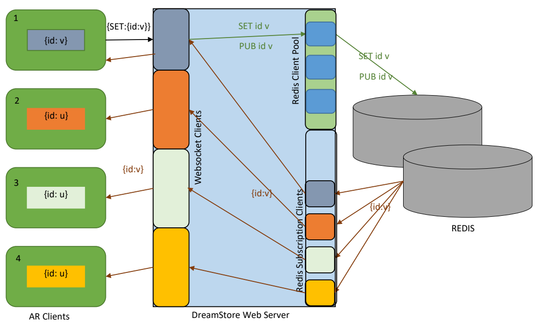


Figure 3: Communication between the ARView management library and the backend infrastructure. The WebSocket clients and the redis subscription clients are color-paired with their AR client applications and AR objects respectively

AR clients in the same shared physical space are required to acknowledge changes by each other while at the same time, ensuring that they can function without explicitly tracking every other client in the system. Decoupling the clients enforces scalability at the abstraction level by allowing them to operate independently of each other. Hence we model the multi-client augmented reality environment as a loosely-coupled system. Publish/subscribe is a widely used interaction paradigm in such environments designed for scalability and real-time data synchronization across components. We use a pub-sub broker on top of the primary key-value store for data synchronization across different AR clients connecting to the DreamStore platform through web socket connections. Although our prototype for the experiment in this paper uses *Redis* for both the key-value store and the pub-sub broker, the system design is not restricted to it and can use any of the other key-value stores and pub-sub brokers.

The WebSocket server creates a redis channel for each object that is queried by the AR clients. It has a dedicated redis subscription client for each AR client connecting to it, which subscribes to updates on every AR object it queries. These updates are published to the AR client through the established WebSocket connection. Figure 3 shows the data synchronization across AR clients in action. A `READ` request on an object not maintained in ARView generates a `READ` request for the backend store. The application server maintains a pool of redis clients to issue `SET`, `GET`, and `PUBLISH` requests to the redis store. Algorithm 2 details the processing of a request generated by the view-management module on the client-side by the application server.

The WebSocket server manages the communication between Redis and the websocket clients. It maintains a unique redis *subscription* client for each AR client connecting to it. The subscription client subscribes to updates on every object in the data-store that is accessed by the corresponding AR client, and forwards these updates as soon as they are available to the WebSocket server, which publishes it to the corresponding AR client's WebSocket channel. We do not differentiate between the different priorities of READ calls in this section for the ease of explanation. We explain how these priorities affect this communication in section 4.2. Each client

```
Function handle_message(client, message, registry):
    operation ⟵ message.key
    if operation is READ then
        object_id ⟵ message.value
        publish fetch_from_store(object_id) on client
        if object_id not in registry[client] then
            add object_id to registry[client]
            while true do
                update ⟵ update on channel object_id
                if update is not None then
                    publish update on client
                end
            end
        end
    end
    else if operation is WRITE then
        object_id ⟵ message.value.key
        object_value ⟵ message.value.value
        publish_to_store(object_id, object_value)
    end

Function publish_to_store(object_id, object_value):
    connection ⟵ connection_pool.get()
    connection.execute("SET object_id object_value")
    connection.execute("PUBLISH object_id {object_id:
      object_value}")

Function fetch_from_store(object_id, object_value):
    connection ⟵ connection_pool.get()
    value ⟵ connection.execute("GET object_id") return
      value
```

Algorithm 2: Request handling by the WebSocket server for the server requests generated by the view-management library on the client side.

maintains a subset of the objects from the store that it is currently operating on in a client-side cache ARView. This view is maintained asynchronously by our client-side library. When a `READ` call is issued on an object which is not present in the local-cache, the library sends a {"`READ`" : object_id} to the server. The WebSocket server maintains a pool of redis-client connections for issuing `SET`, `GET`, and `PUBLISH` commands to Redis. On receiving a `GET` message from a client, the server uses a connection from the pool to fetch the queried object and sends it to the requesting client. When a client issues a `GET` on an object, the corresponding redis subscription client creates a subscription on a channel name corresponding to its object id. The WebSocket connections continuously listen for updates on the redis subscription clients corresponding to them, every new message on this connection is immediately sent back to the client-device where the view-management library asynchronously updates the local copy of the corresponding object. When the `COMMIT` method is invoked, the view-management library sends a {"`WRITE`" : {"object_id" : object_value}} for each object with a set dirty bit. The server issues a `SET` and a `PUBLISH` command to redis on a connection from its pool of redis clients. The PUBLISH command is issued on the channel named – object id with the message {"object_id" : object value}. Thus the synchronization between multiple clients is managed without any effort from the application developer.

7 Evaluation

Existing AR research is lacking in performance evaluation of large-scale shared reality experience deployments [32]. User evaluations focused on human perception and cognition in AR, user-performance, and user-interaction is important for evaluating AR applications. However, a user-study of this nature is impractical for studying system performance with hundreds of users accessing the same shared AR space. DreamStore evaluation utilizes query-intensive workloads emulated from realistic AR application profiles.

7.1 Workload Generation

Because of a lack of open location-tagged AR workloads, we used an indoor localization dataset (referred to as *IPS* in the text) published by Mohammadi et al. [20] to simulate AR application sessions. This dataset is generated from a grid of iBeacons deployed in a campus library (200 ft x 180 ft). We utilize the data used for localization training that maps a time-stamped measurement of multiple beacon reading to a grid in the physical space. These grids are analogous to the regions in DreamStore (Section 5). In order to emulate multiple client workloads, we segmented the 1420 point dataset into 14 distinct user sessions separated by physical space and a significant time difference between the start of the sessions. This provides us a realistic time-stamped human-movement path data. In order to emulate an AR scenario, we designated ten unique AR objects to each grid and added a `READ` request at each data point for 1 to 5 objects at random. Each point in the segmented user-session is annotated with the timestamp it was generated at, the unique region it is located in, and an assigned set of AR object identifiers. The typical time gap between consecutive measurements between two consecutive points in each user session is between 2-4 seconds, making these sessions representative of the reduced query-workload one would expect after applying query-thresholding (Section 4.2). We simulated 1000 user sessions by picking a random segmented user-session for each user, introducing a random start delay, and assigning randomly generated `READ` requests picking from the objects assigned to the grid the point belongs to. Each simulated user either uses the same succession of points as in the segmented user-sessions obtained from the training dataset or inverts it to emulate a new path resulting in a total of 28 distinct paths.

In order to emulate a generic AR application, we created an Android application powered by DreamStore that scans the space for identifiable markers for which it has associated object information in the database. We placed twelve unique fiducial markers at different locations in an office cubicle, each identifying an object in the physical space, e.g., computer screen, bulletin board, appliances, etc. The data stores were populated with synthetic information associated with these objects and tagged with geographical coordinates. A read workload was created by scanning the space for about four minutes, querying information on one or more objects at a time, while recording all the API calls. The sequence of `READ` calls in the recorded workload was permuted to create 1000 different client sessions (referred to as *Generic* workload in the text). Unlike the *IPS* workload, this usage scenario investigates a small physical space, where the user would be free to scan objects in any order without any physical displacement. Hence permuting the `READ` calls in any order still maintains the realism of the workload. To test the performance of the system under updates, we generated a *Generic-MIXED* workload by introducing about 5% (of the total `READ`s in the workload) random updates in the workload on one of the objects from the current read set at the timestamp right before the introduced update call. The think-time for each update varies from fifteen to forty seconds. We simulated these 100 clients simultaneously for both the read and the mixed workloads on different test setups. All `READ`s are set to `HIGH` priority, i.e., they always render the queried objects in full detail. For the updates, we treat all `WRITE`s as `COMMIT`s, causing the clients to trigger a `SET` call to the backend on each update.

7.2 Test Setup

The data platform (Redis) and the WebSocket server were set up on an n1-highmem-4 Google Cloud instance (4 vCPUs, 26GB RAM). The client workloads were divided evenly over two different n1-standard-4 Google cloud instances (4 vCPUs, 15GB RAM) for each test scenario.

We evaluated the following test configurations for the three workloads – *IPS*, *Generic*, and *Generic-MIXED*.

- **Naive** – In this setup, each DreamStore API call translates to a server call, triggering a backend request for each implicit or explicit data interaction on the UI.
- **Reduced** – In this setup, we utilize the query thresholding described in Section 4.2 to reduce the generated `READ` calls to the backend.
- **DreamStore** – The clients cache objects on top of the query reduction in the previous setup. For the *Generic* and *Generic-MIXED* workloads, the clients cache an object the first time it is accessed and maintain it in the cache throughout the rest of the session. For the *IPS* workload, the objects are prefetched into the cache according to a directional prefetching strategy, evicting the least recently cached objects when the cache is full.

7.3 Performance Metrics

We use metrics that are useful for evaluating interactive data exploration systems [24], in addition to metrics that are performance indicators for specific DreamStore features.

- **Query Issue Rate**: The number of queries issued to either the local cache or the backend, indicating the efficacy of the interaction mapper in reducing the query workload through thresholding and prioritization.
- **Latency/Response Time**: The response time on the client for each `READ` and `WRITE` request.
- **Update Propagation Time**: The time for the object value update from `COMMIT` by a client to propagate to other clients maintaining the object in their local cache.
- **Cache Hit Ratio**: The proportion of `READ` requests serviced by the local client cache, indicating the effectiveness of the prefetching policy.
- **Interactive Constraint Violation**: The proportion of requests for which the response time exceeds 100ms. Requests exceeding this threshold would potentially hamper the user experience.

7.4 Results

Due to the time granularity (at least 2 seconds) of recording in the dataset that *IPS* workload emulates, it resembles the reduced workload that would be generated by the DreamStore query reduction methods. Hence, we do not present the workload reduction evaluation metric (*query issue rate*) for it. Additionally, since it is a `READ`-only workload, *update propagation time* is not an applicable metric for its evaluation.

7.4.1 Generic Workload

We use value thresholding (Section 4.2) to reduce the query workload generated by the DreamStore clients, issuing `READ` calls only for objects which are queried at least twice in the `READ`-aggregation period – 100 ms.

Query Issue Rate: Figure 4 and Table 1 show the reduction in query-rate for a client session from about 5 every 100ms to close to 1 in the *Generic* workload after thresholding.

Average Latency: The reduction in query-rate dramatically reduces the workload on the server, improving the average response time per query by about 4 times, as can be seen in the difference between the Naive and the Reduced setup in Table 1.

Interactive Constraint Violation: Table 2 shows the proportion of requests that cross the interactive latency threshold of 100ms. Value thresholding brings down the violation rate from 0.8 to 0.07, which is reduced to almost 0 by enabling local cache in the DreamStore setup. Figure 5a and Figure 5b show the response time per query for one of the client sessions in the *Generic* workload.

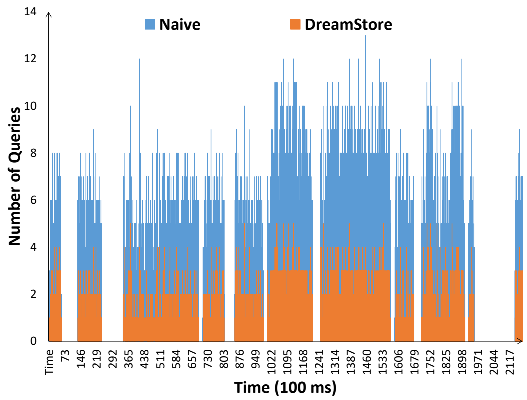


Figure 4: Query Issue Rate – Number of `READ` queries issued per 100 ms during a client session of *Generic* workload under the Naive and DreamStore setups

Table 1: Aggregate `READ` performance measures and generated *query issue rate* for the Naive, Reduced (RED), and DreamStore (DS) setups. – *Generic* workload

Measure \ Setup		NAIVE	RED	DS
Response Time (ms)	AVG	300.2	72.54	8.7
	MED	242	69	9
	MIN	53	46	3
	MAX	1290	239	124
	SD	189.5	20.82	5.2
Query Rate/client (queries/100 ms)	AVG	4.24	1.3	1.3
	MED	5	1	1
	MIN	0	0	0
	MAX	13	5	5
	SD	3.09	1.17	1.17

Table 2: Latency Constraint Violation (LCV) – The ratio of requests exceeding the 100 ms latency mark for the different test setups under *Generic* workload

Setup	Requests/Client	LCV
Naive	9239	0.8
Reduced	2849	0.073
DreamStore	2849	8.54×10^{-6}

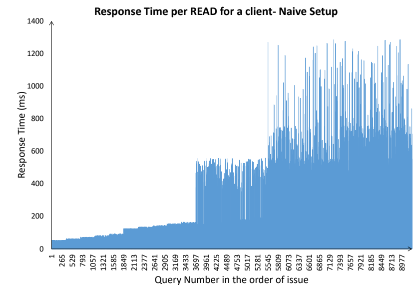
7.4.2 Generic-MIXED Workload

The *Generic-MIXED* workload shows performance characteristics similar to that of the *Generic* workload.
Average Latency: Table 3 shows the response time statistics for the different test setups. The `READ` response times are not affected by the occasional updates in the workload.

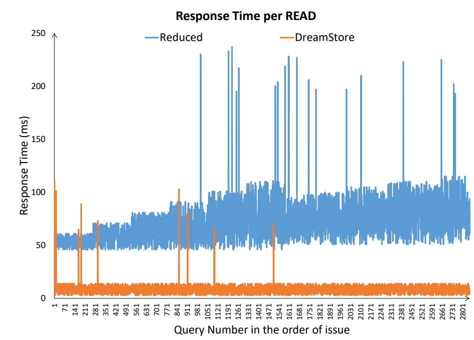
Table 3: Aggregate `READ` performance measures for the Naive, Reduced, and DreamStore setups. – *Generic-MIXED* workload

Setup	AVG	MED	MIN	MAX	SD
Naive	303.56	166	51	1323	235.17
Reduced	82.9	75	52	251	28.12
DreamStore	8.97	9	4	183	6.52

Update Propagation Time: The DreamStore setup measures update propagation time by including the update time in the object value when a client issues a `COMMIT` call on it. Table 4 shows the updates are propagated to all the clients in a reasonable time, well within the interactive latency constraint. Moreover, the propagation times are consistent across the workload with little variation.



(a) Response time per query for Naive test setup – *Generic* workload



(b) Response time per query for Reduced and DreamStore test setups – *Generic* workload

Figure 5: Response time per query (plotted in the issue-order) for one of the simulated clients in the *Generic* workload.

Table 4: Update Propagation Time for *Generic-MIXED* workload

AVG	MED	MIN	MAX	SD
78.9	85	69	91	4.25

7.4.3 IPS Workload

The *IPS* workload has a wide variation in the client-session running times ranging from 18 seconds to 2972 seconds or about 50 minutes. We report the latency statistics and the cache hit rate over the entire workload.
Cache Hit Rate: Each data point in a client-session has a physical zone assigned to it, identified by a letter and a number combination, positioning it in a unique cell in the grid. We implement a directional prefetching strategy as described in Section 5, and achieve a cache hit rate of 0.91 over the entire workload.
Average Latency: Table 5 shows the aggregate query response time statistics for the *IPS* workload. As explained earlier, the *IPS* workload does not present any avenue for query reduction because of the way it is designed. The performance improvement seen in the DreamStore setup over the Naive setup comes from the directional prefetching, which ensure over 90% requests are serviced by the local cache for each client.

The Naive setup for the *IPS* workloads corresponds to the Reduced setup for the *Generic* workload in terms of the order of query

Table 5: Aggregate READ performance measures for the Naive and DreamStore setups. – *IPS* workload

Setup	AVG	MED	MIN	MAX	SD
Naive	89.1		71	49	235
DreamStore	11.25	12	5	176	3.52

issue rate, and both show similar performance. The latency constraint violation rate is about 0.11 for the Naive setup, showing more than 10% of the queries exceeding the interactive latency threshold. The *IPS* workload achieves a latency constraint violation rate of about 0.01 with the DreamStore setup from the performance gains due to prefetching and caching.

8 Discussion

In contrast to the specialized data platforms for AR discussed in section 2, DreamStore strips down the amount of information it needs at the backend by having a dedicated store for virtual object properties and relying on a different visual recognition database for marking the placement of these objects in space. The database is not required to store spatial information for entire spaces such as the large scene graph in the system by Schmalteig et al [27], making the search and storage more efficient. DreamStore can be implemented along with platforms like Spatial Anchors – utilizing its persistent object tracking and storage, and implementing the local cache management techniques on top of it.

Existing works on AR architecture and shared AR do not focus on the problem of data synchronization in a multi-client environment, and are geared towards placing and tracking AR objects in the shared world view between different clients. DreamStore can enhance these applications with the proposed workload reduction techniques and prefetching and caching strategies.

Query Workload Reduction: Reducing the query workload generated by the clients through query prioritization and thresholding (Section 4.2) improves the average latency by about 4 times in the *Generic* (Table 1) and *Generic-Mixed* (Table 3) workloads. Some of the reduction in the READ requests in the *Generic-MIXED* workload can be attributed to the WRITE requests generated by the clients as the READs are suppressed for the think-time duration of an active WRITE request. The latency per query with the reduced setup does not show as wide a spread in the naive setup ensuring a more consistent system response throughout the session. The response time gradually keeps increasing for the naive setup with sharp changes, possibly indicating connection pool recycling or WebSocket connection backlog issues because of increasing system load (Figure 5). The *Reduced* setup shows a gradual increase in the response time across the session with few occasional spikes that are not as deviant from the latency distribution across the session as seen in the Naive setup.

On account of how the *IPS* workload is designed (course granularity of location logging), the query distribution resembles the workload obtained after reduction, and hence the performance improvement seen in the DreamStore setup for *IPS* can be attributed completely to the prefetching and caching strategies.

Prefetching and Caching: The *Generic* and *Generic-MIXED* workloads access a small number of objects over the entire session that can all be cached on each client. Hence, the *cache hit rate* is not a relevant metric for the two workloads, as they would only have a cache *miss* for an object the first time it is accessed in the session. The DreamStore setup shows a consistent response time across the session 5, with the spikes in the query latency indicating the first time a new object is accessed, and the *READ* call is serviced from the backend data store.

A high cache hit rate (0.91) with the *IPS* workload shows the effectiveness of even a relatively simple prefetching strategy. The cache misses can be attributed to the obvious misses at the start of each client session, and a few points in some of the client sessions where the object access pattern is not in line with the directional prefetching strategy. Some of these measurements skip cells in the grid along with an unusual jump in time, indicating possible missing measurements in the base dataset. The deployed IPS system and the ease of predictability of user movement pattern would impact the effectiveness of the emloyed prefetching strategy.

8.1 Limitations

Although DreamStore focuses on reducing visual clutter along with providing consistent and interactivity latency, there are multiple design considerations from the perspective of human cognition and perception for AR visualization that were not considered. An important issue that arises from this work would be visualizing updates on objects. To a user interacting and manipulating an AR object, any explicit updates issued by them are something that they would expect a feedback for. However, system-initiated updates (updates in the underlying data due to source update or an action from another user) can potentially distract users.

Intuitively, certain objects and tasks should prefer the DreamStore style of pushing every object update to the interface. For other objects and tasks, it could be useful to suppress the updates if the user's attention is required elsewhere and the task is unaffected by the update in consideration, or emphasize the update by calling out users attention if that is more helpful to the user task and performance. Another potential problem with the frequency of visualized updates in DreamStore could be the system ignoring objects because of their low visual query frequency, even though the user might be interested in them.

9 Conclusion and Future Work

Through performance evaluation of query-intensive workloads that emulate multiple AR clients in a shared physical space, we showed that DreamStore can reduce effective query-rate through workload-reduction by about 5 times, and along with client-side caching, can bring down the average response time by an order of magnitude compared to the baseline setup. DreamStore can effectively ensure interactive latency, with an interactive constraint violation rate of almost zero compared to 80% in the baseline setup, and provide near-real-time update propagation across clients well within the 100 ms interactive latency threshold. We show the effectiveness of an appropriate prefetching strategy, achieving a cache hit ratio of 0.91 with a directional prefetching strategy for an indoor movement workload, improving the average query latency by an order of magnitude.

DreamStore can enchance shared AR experiences in a variety of settings such as shared AR gaming experiences and collaborative work environments [4, 28], by effectively mapping user interactions in AR to a reduced set of data queries along with priority such that they do not overload the data platform, and enabling easy and rapid development of applications with interactive latency under updates for a smooth AR experiences with numerous clients. It can power AR data analytics use-cases [5, 17] by facilitating real-word data querying in AR by providing a backend datastore and a framework for work-sharing and data synchronization between multiple clients.

In the future, we intend to release client-environment specific view-management libraries for popular AR platforms. We are also working on further refining the interaction module and defining the interaction mapper for a broader set of user actions for different interaction modalities.

10 Acknowledgement

We acknowledge the support of the U.S. National Science Foundation under the awards IIS-1910356 and CAREER IIS-1453582.

REFERENCES

[1] R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, and B. MacIntyre. Recent advances in Augmented Reality. *IEEE computer graphics and applications*, 21(6):34–47, 2001.

[2] R. T. Azuma. A survey of Augmented Reality. *Presence: Teleoperators and virtual environments*, 6(4):355–385, 1997.

[3] J. Bernardes, R. Tori, R. Nakamura, D. Calife, and A. Tomoyose. Augmented Reality games. *Extending Experiences: Structure, analysis and design of computer game player experience*, 1:228–246, 2008.

[4] P. Bhattacharyya, Y. Jo, K. Jadhav, R. Nath, and J. Hammer. Brick: A Synchronous Multiplayer Augmented Reality Game for Mobile Phones. CHI EA '19, page 1–4, New York, NY, USA, 2019. Association for Computing Machinery.

[5] C. Burley and A. Nandi. ARQuery: Hallucinating Analytics over Real-World data using Augmented Reality. *CIDR*, 2019.

[6] Q. Dong. Skip the line: Restaurant wait times on Search and Maps. *Google Blog*, 2017.

[7] R. Ebenstein, N. Kamat, and A. Nandi. FluxQuery: An Execution Framework for Highly Interactive Query Workloads. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*. ACM, 2016.

[8] H. Fuchs, M. A. Livingston, R. Raskar, K. Keller, J. R. Crawford, P. Rademacher, S. H. Drake, A. A. Meyer, et al. Augmented Reality visualization for laparoscopic surgery. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 934–943. Springer, 1998.

[9] A. Gallala, B. Hichri, and P. Plapper. Survey: The Evolution of the Usage of Augmented Reality in Industry 4.0. *IOP Conference Series: Materials Science and Engineering*, 521:012017, may 2019.

[10] P. Geiger, M. Schickler, R. Pryss, J. Schobel, and M. Reichert. Location-based mobile Augmented Reality applications: Challenges, examples, lessons learned. 2014.

[11] E. Gelenbe and F.-J. Wu. Future research on cyber-physical emergency management systems. *Future Internet*, 5(3):336–354, 2013.

[12] D. Grosu, A. T. Chronopoulos, and M.-Y. Leung. Load balancing in distributed systems: An approach using cooperative games. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 10–pp. IEEE, 2001.

[13] A. Guo, I. Canberk, H. Murphy, A. Monroy-Hernández, and R. Vaish. Blocks: Collaborative and Persistent Augmented Reality Experiences. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 3(3), Sept. 2019.

[14] B. Haynes, A. Mazumdar, A. Alaghi, M. Balazinska, L. Ceze, and A. Cheung. LightDB: a DBMS for virtual reality video. *Proceedings of the VLDB Endowment*, 11(10):1192–1205, 2018.

[15] A. Henrysson and M. Ollila. UMAR: Ubiquitous mobile augmented reality. In *Proceedings of the 3rd international conference on Mobile and ubiquitous multimedia*, pages 41–45. ACM, 2004.

[16] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson. Informed mobile prefetching. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 155–168, 2012.

[17] M. Khan. AR Heatmap. `https://go.osu.edu/heatmap`.

[18] C. D. Kounavis, A. E. Kasimati, and E. D. Zamani. Enhancing the tourism experience through mobile augmented reality: Challenges and prospects. *International Journal of Engineering Business Management*, 4:10, 2012.

[19] Microsoft. Spatial Anchors. `https://azure.microsoft.com/en-us/services/spatial-anchors/`.

[20] M. Mohammadi, A. Al-Fuqaha, M. Guizani, and J. S. Oh. Semi-supervised Deep Reinforcement Learning in Support of IoT and Smart City Services. *IEEE Internet of Things Journal*, pages 1–12, 2017.

[21] D. Nieklas and B. Mitschang. A model-based, open architecture for mobile, spatially aware applications. In *OOIS 2001*, pages 392–401. Springer, 2001.

[22] Pantomime. Pantomime creatures. `http://pantomimecorp.com/pantomime-creatures/`.

[23] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *the VLDB Journal*, 10(4):334–350, 2001.

[24] P. Rahman, L. Jiang, and A. Nandi. Evaluating interactive data systems. *The VLDB Journal*, 29(1):119–146, 2020.

[25] R. Rosenholtz, Y. Li, and L. Nakano. Measuring visual clutter. *Journal of vision*, 7(2):17–17, 2007.

[26] C. Sandor, M. Fuchs, A. Cassinelli, H. Li, R. Newcombe, G. Yamamoto, and S. Feiner. Breaking the barriers to true augmented reality. *arXiv preprint arXiv:1512.05471*, 2015.

[27] D. Schmalstieg, G. Schall, D. Wagner, I. Barakonyi, G. Reitmayr, J. Newman, and F. Ledermann. Managing complex augmented reality models. *IEEE Computer Graphics and Applications*, 27(4), 2007.

[28] M. Sereno, X. Wang, L. Besancon, M. J. Mcguffin, and T. Isenberg. Collaborative Work in Augmented Reality: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 2020.

[29] M. Singh and M. P. Singh. Augmented reality interfaces. *IEEE Internet Computing*, 17(6):66–70, 2013.

[30] H. Slay, B. Thomas, and R. Vernik. Tangible User Interaction Using Augmented Reality. *Aust. Comput. Sci. Commun.*, 24(4):13–20, Jan. 2002.

[31] R. Szeliski. *Computer Vision: Algorithms and Applications*. Springer Science & Business Media, 2010.

[32] A. Tang, C. Owen, F. Biocca, and W. Mou. *Performance Evaluation of Augmented Reality for Directed Assembly*, pages 311–331. Springer London, London, 2004.

[33] H. Tramberend. Avocado: A distributed virtual reality framework. In *Virtual Reality, 1999. Proceedings., IEEE*, pages 14–21. IEEE, 1999.

[34] G. R. Vesto. Augmented reality enhanced triage systems and methods for emergency medical services, Aug. 5 2011. US Patent App. 13/204,524.

[35] A. Webster, S. Feiner, B. MacIntyre, W. Massie, and T. Krueger. Augmented reality in architectural construction, inspection and renovation. In *Proc. ASCE Third Congress on Computing in Civil Engineering*, pages 913–919, 1996.

[36] E. Zhu, A. Hadadgar, I. Masiello, and N. Zary. Augmented reality in healthcare education: an integrative review. *PeerJ*, 2:e469, 2014.