MultiplayAR

Senior Design - Computer & Information Science
Spring 2018

Akshat Agrawal
Krishna Bharathala
Devesh Dayal
Jacob Kahn
Amit Patel

Professor CJ Taylor
TA Judy Weng

Executive Summary

Through novel use of computer vision algorithms, we built a pluggable framework that solves the challenges of building shared multiplayer augmented reality experiences, allowing developers to focus entirely on creating immersive experiences.

Overview of Problem

Augmented Reality (AR) uses computer vision techniques to overlay computer-generated 3D-assets onto a live camera feed and has experienced a surge of interest recently. To facilitate AR application development, Apple and Google released suites of developer tools, last Fall (ARKit and ARCore, respectively). Notably, multiplayer functionality was not included within these suites.

Multiplayer AR experiences exist today, but they require significant technical overhead to set-up and often necessitate user calibration. At the core of multiplayer AR system's technical challenges is the synchronization of each device's AR-environment with respect to a shared one, such that each user's interactions in real-world space appear consistent across all players.

Market and Competition

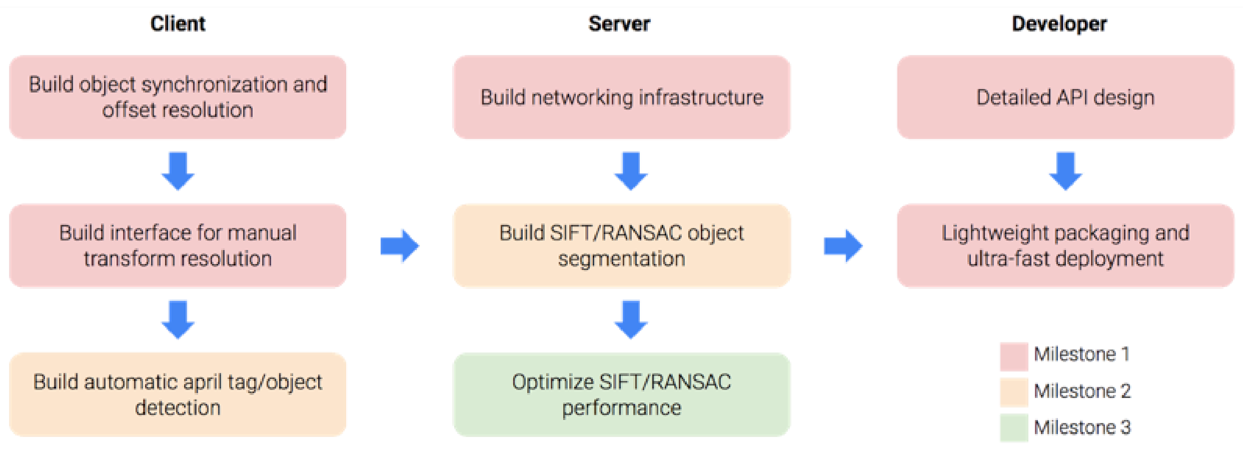
There are currently 600 million devices in the United States that have augmented reality capabilities (500 million iPhones 6S and up, and 100 million Android phones). These users represent a growing market interested in augmented reality applications. Mobile Gaming was a $45B market in 2017, with 19% YoY growth since 2015.

The first mainstream application of augmented reality was Pokemon Go which was released in the summer of 2016. The game was quick hit garnering 50 million daily users in the first week it was out. Since then, many companies have tried to emulate the virality of augmented reality games.

Right now there are a few companies working on multiplayer augmented applications: Google, Apple, Snapchat, Vuforia, WingnutAR, to name a few. However, none of these groups are currently working on a way to let ***others*** build multiplayer augmented reality solutions. Our goal is to try and help developers and designers create their own applications and compete against the players that are currently in the space.

Implementation

Overview
To synchronize 3D-worlds across devices, we first capture images from each device. Employing custom algorithms built on computer vision techniques, we detect objects present across images, but shifted or rotated from different devices' points-of-view. Computing the transformation from one device's point-of-view to that of another allows us to determine their relative displacements in 3D-space. This information allows us to anchor a coordinate system for a virtual, shared AR world.


Client
Server
Developer
Build object synchronization and offset resolution
Build networking infrastructure
Detailed API design
Build interface for manual transform resolution
Build SIFT/RANSAC object segmentation
Lightweight packaging and ultra-fast deployment
Build automatic april tag/object detection
Optimize SIFT/RANSAC performance
Milestone 1
Milestone 2
Milestone 3

We approached the problem by splitting up the solution into milestones. Upon completion, each milestone would allow us to test a full end-to-end solution of the problem with different levels of difficulty and complexity. The first milestone was a recreation of manual resolution and object synchronization across a network. The second milestone added the first version of computer vision into our pipeline. The third milestone optimized our evaluation parameters over our existing milestone two solution.

Our project can broken up into three sections as labeled above: client, server, and developer.

Client Side

On the client side, we implemented a framework that sends camera captures to the backend server to begin a calibration setup routine. Along with raw pixels, individual devices initialize ARKit tracking, record any detected feature points and package them with the image payload. Once the server replies with a set of candidate anchor points, a new world origin point is set and any number of multiplayer objects can then be added into the scene. Their position relative to the anchor is then continuously broadcasted to all other connected clients whenever their world coordinates change at a configurable frame rate.

Feature Point Collection
Periodically, ARKit reports 3D coordinates of features in the tracked world that it has a reasonable distance estimate for. These points are reported as events and our framework consumes these events over a brief window of time and are filtered based on the camera's movement in physical space.

Position Syncing
Each device assigns itself a unique ID (validated by the server) and then polls the server for the state of the world at a configurable interval rate. The state of the world is compared against the local state of the device and new information is patched in. Each device can add any number of new game objects to their scenes and our scripts capture their 3D world position and broadcast them to the server periodically to update the global state of the world. Low-latency overhead is maintained by sending across only float vector information. Assets are generated and rendered on the device itself.

Server Side

On the server side, we implemented an algorithm that combines computer vision with AR point data from our clients to calculate possible anchor points. The server then sends the candidate anchors back to one of the clients, where the user will choose the appropriate anchor. Once the anchor has been chosen, the server syncs the universe of objects for both clients.

Candidate Anchor Generation Procedure:
Inputs: Both clients send the server an image and a list of AR points generated by ARKit. The AR points are points of high interest according to ARKit, and are importantly points that the phone can locate in 3D Space.

Algorithm: First, we use OpenCV Contrib's implementation of SIFT to extract keypoints in each image and description vectors for each of them (referred to as descriptors). We then use a randomized FLANN KD Tree implementation from OpenCV to calculate matching scores between keypoints using their descriptors. Specifically, we implemented the ratio test described by David Lowe in his paper. We then filter for the best matching scores. Now, we augment our SIFT matching scores with our AR points by weighting each matching score by its "centrality score":

the total distance from the matched keypoints in each image to the k closest AR points to them. This way, we upweight matchings between points that are very close to AR points in the images. We then use the top 4 points by centrality score to compute a homography transform and we store them. Finally, we return to the client the 4 AR points closest to each of the 4 top ranked SIFT keypoint matches as potential anchor points.

Syncing:
Once the anchor has been chosen for one client, the server uses original homography to send the other client the same anchor. Now, both clients can register objects with their 3D offsets relative to the anchor point. The server then accepts updates to the world of objects and uses a sync endpoint to keep every client up to date.

Server Architecture and Details: We wrote our server in C++, using the crow web framework. The server communicates with clients using http routes.

Developer Side

On the developer side, we wanted to create a seamless experience in getting our framework up and running on Unity.

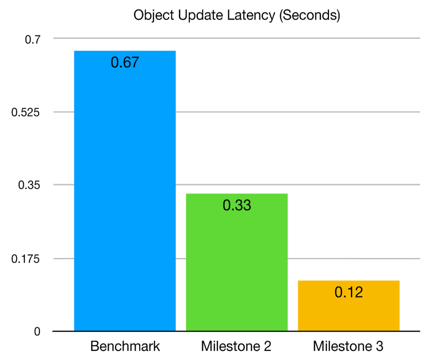
To that end we spent a lot of time brainstorming the best ways to integrate the framework into our user's existing solutions. The best way to go about doing this was through Unity scripts, which can be attached to Unity Objects by just dragging and dropping modular components to game objects. These can be baked into prefabs which are reusable across multiple projects, making the entire development experience as effortless as possible.

In addition, we did not want to burden the user with having to set up their own server, especially for designers who may not have technical experience. To solve this challenge, we added a "One-Click Deploy" button to our documentation/tutorial website. This button automatically deploys a server on Docker which can be added to our framework by just adding a URL parameter to the Unity sidebar.

Evaluation

We decided on three main criteria to evaluate our framework on:

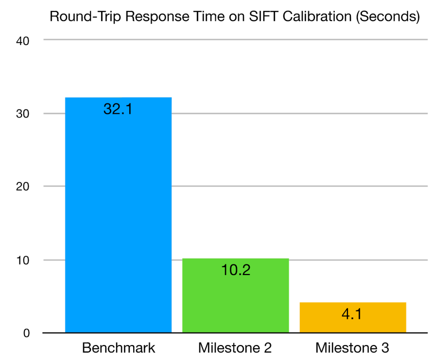
1) Calibration



Length of time it takes for tracking to begin.

Once the user opens the application and approves the use of the camera, we want to reduce the amount of time it takes until they can use the app. Our goal would be to keep this under ten seconds. This is broken up into five seconds to detect the plane and place the anchor, and then another five seconds to transmit the rotation.

2) Tracking



Reduce the latency the system generates

Along with tracking we want to make sure that the transform across the network doesn't take up too much time which could cause lag between objects on various screens, leading to suboptimal game-play. Eventually our goal should be to reduce this number to the speed of sending data over the network.

3) Ease of Use

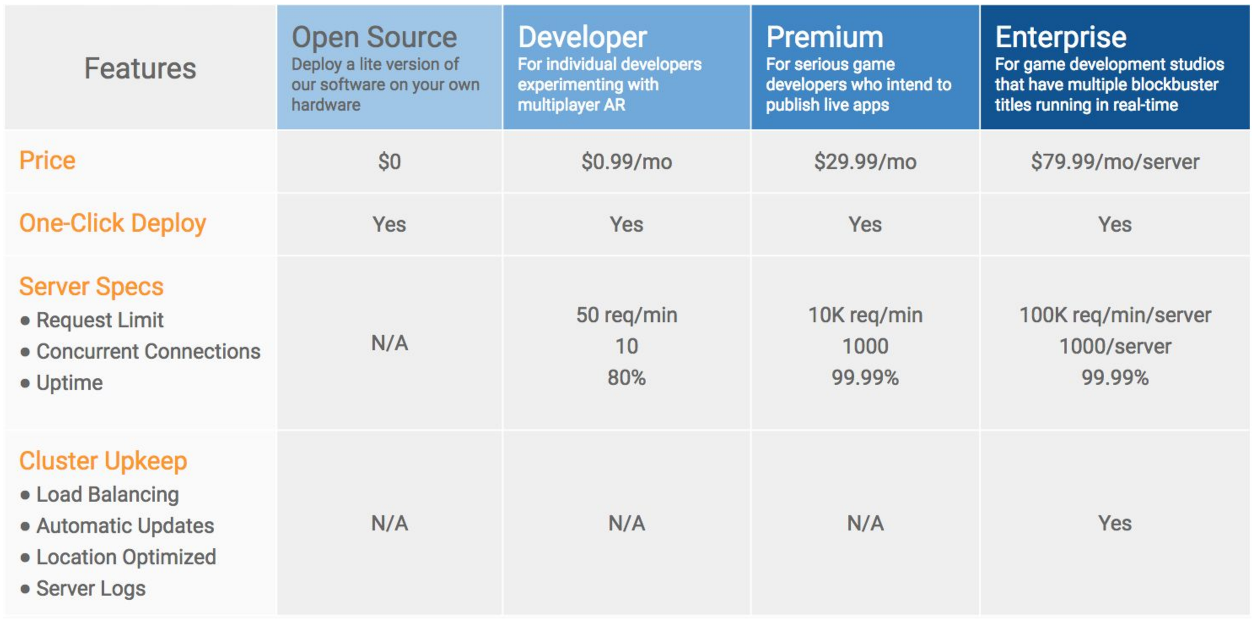
Limit the set-up overhead in the development workflow.

This means reducing the amount of lines it takes to integrate our solution. Ideally, the user would just have to use one import statement and then use a MultiplayerGameObject constructor and the rest would be taken care of.

Time (s) required for ARKit initialization time	1.1	Latency (s) between local update and peer update	0.12	Lines of code needed to integrate functionality	0
Time (s) required for server to return anchors	4.1			Lines of code needed to set up server instance	0

Additionally, you should also identify threats to validity, i.e. assumptions you've made and other factors that may indicate that your results do not actually demonstrate what you've set out to prove. Last, also describe any legal and ethical considerations: in what ways could you, or end-users, or society in general, be put at risk through the use of your solution?

Revenue Model and Costs



Features	Open Source Deploy a lite version of our software on your own hardware	Developer For individual developers experimenting with multiplayer AR	Premium For serious game developers who intend to publish live apps	Enterprise For game development studios that have multiple blockbuster titles running in real-time
Price	$0	$0.99/mo	$29.99/mo	$79.99/mo/server
One-Click Deploy	Yes	Yes	Yes	Yes
Server Specs ● Request Limit ● Concurrent Connections ● Uptime	N/A	50 req/min 10 80%	10K req/min 1000 99.99%	100K req/min/server 1000/server 99.99%
Cluster Upkeep ● Load Balancing ● Automatic Updates ● Location Optimized ● Server Logs	N/A	N/A	N/A	Yes

We opted to create a business model based on "Platform as a Service" products. Similar to products like AWS, Firebase, and Heroku we created a tiered pricing model to allow users to choose which pricing is best for their use case. In this way we can provide the best service for our best customers while also maintaining our mission of keeping the software accessible to all.

The long term goals for this type of model are to transition users from lower tiers to higher tiers. The two main transitions we want to focus on are Open Source ⟶ Developers and Developers ⟶ Premium. The first transition, Open Source to Developers costs only the marginal amount for us to host the user's servers for them. For the user, this reduces the headache they might have of having to maintain their own hardware and server instances. The second transition from Develop ⟶ Premium represents a serious upgrade from hobby apps to live apps. As we continue to develop this platform, we expect to add more tooling that will help with the publishing and deployment of quality applications and incentivize our users to move to the premium tier to capture these benefits.

Costs

Outside of the cost of hosting servers for our users, the PAAS does not incur any product costs. Based on our tiered pricing model, we expect all server costs to be offset by the user's payment. All other revenue generated from users will be used to offset non-technical costs like salary and sales/advertising.

Individual Contributions

Documentation and Usability -- Krishna

Server Infrastructure -- Jacob and Amit

Unity Framework -- Akshat and Devesh