Google Street View as a Medium for Social Gaming

Steven Dao
Tien Ho
Austin Pahl

Follow this and additional works at: https://scholarcommons.sc.edu/senior_theses

Part of the Computer Sciences Commons

Recommended Citation
Dao, Steven; Ho, Tien; and Pahl, Austin, "Google Street View as a Medium for Social Gaming" (2017). Senior Theses. 145.
https://scholarcommons.sc.edu/senior_theses/145

This Thesis is brought to you for free and open access by the Honors College at Scholar Commons. It has been accepted for inclusion in Senior Theses by an authorized administrator of Scholar Commons. For more information, please contact dillarda@mailbox.sc.edu.
GOOGLE STREET VIEW AS A MEDIUM FOR SOCIAL GAMING

By

Steven Dao
Tien Ho
Austin Pahl

Submitted in Partial Fulfillment
of the Requirements for
Graduation with Honors from the
South Carolina Honors College

May, 2017

Approved:

Dr. Jose M. Vidal
Director of Thesis

Steve Lynn, Dean
For South Carolina Honors College
# Table of Contents

**Thesis Summary**  
4

**Introduction**  
4

**Planning**  
5
  - Waterfall Development Process  
  - Technologies  
    - Collaboration/Communication Tools  
    - Frontend Frameworks  
      - Angular 2  
      - AngularFire 2  
      - Bootstrap  
      - HowlerJS  
      - Google Maps API  
    - Backend Frameworks  
      - Firebase  
      - Django  
      - MEAN Stack  
  - Frontend and Backend Overview  

**Design**  
10
  - Mockups  
  - Architecture  

**Implementation**  
15
  - Architecture  
  - Services  
  - Components  
  - UML Diagram  
  - Challenges/Issues  

**Validation**  
18
  - Testing  
  - Debugging  

**Results**  
19
  - Finished App  
  - Presentations  

**References**  
22

**Appendix: Tools and Technologies Used**  
23
Thesis Summary

In this thesis project, we planned and built a multiplayer web game called safehouse from Fall 2016 to Spring 2017. We were part of a larger Computer Science Capstone team: Steven Dao, Tien Ho, Blakeley Hoffman, Brandon Hostetter, and Austin Pahl. We began with a planning phase that resulted in an architecture document (a “blueprint” for the structure of our code) and a requirements document (a list of features that we intended to implement). We showcased a proof-of-concept in December, then released the final version in April. The game is available to play at https://thesafe.house.

Introduction

safehouse is a hide-and-seek variant set in Google Street View that can be played in a web browser with friends or strangers online. Google Street View is a powerful interactive platform with a massive database that includes views on every continent (Google Street View). There are some existing games set in Google Street View, such as Anton Wallén’s GeoGuessr (GeoGuessr) and Nemesys Games’ Pursued (Pursued). In both games, players are placed into a random Street View somewhere in the world and expected to use clues in their surroundings, such as signs, nature, and architecture, to guess where they are located. There are also other experiments that build on Google Street View, such as Einar Öberg’s Urban Jungle Street View (Urban Jungle Street View) which applies post-processing to any photosphere from Google Street View to appear as a jungle. However, there have been few full-scale Street-View-based games that have attracted significant attention since the technology was launched in the U.S. in 2007.

Our Computer Science Capstone group drew inspiration from GeoGuessr and Pursued when we developed the initial concept for safehouse in August 2016. The team consisted of Steven Dao, Tien Ho, Blakeley Hoffman, Brandon Hostetter, and Austin Pahl. We wanted to design a game that was social by design to draw a different audience than GeoGuessr and Pursued, which are both optimized for a solo experience. We decided that hide-and-seek would be a good basis for our game: it is widely known and easy to pick up, and it can fit nicely into Street Views for cities because of the maze of interlocking streets that the players can navigate. The game also “ports” nicely to the Street View format, as players can move around the city much like people would on foot.

After a few weeks of discussion on the premise we had devised and exploring other project ideas, we submitted the following project statement as a proposal for our Capstone:

safehouse is a head-to-head Street View game drawing inspiration from GeoGuessr and Pursued. It will be available to play in a web browser on our website, http://thesafe.house. The game aims to let players discover new places in the world in a uniquely competitive way. At the start of each match, players will find themselves in a random Street View location in an unknown area, pitted against each other with one of two tasks: the pursued must locate a safe house somewhere in the city, and the pursuer must catch the pursued. Each player will have to rely on quick wits, attention to detail, and a good sense of direction in order to succeed.
The proposal was accepted by our Thesis Director and Course Instructor, Dr. Jose M. Vidal. We spent the next nine weeks analyzing our potential audience, designing the UI/UX, defining requirements, and planning the architecture of our app. “Personas,” fictional characters that represent different users that we expect to see, were created and used as a starting point for future decision making. A design was hand drawn and then converted to a Balsamiq wireframe.

We broke ground on coding with our first commit on October 25, 2016, when Brandon Hostetter initialized the Angular project and pushed it to our repository on GitHub. We presented a demonstration of a working prototype on December 6, 2017. After another semester of steady development on the application, we launched version 1.0 on April 24, 2017. The game is currently live and available to play at our website, http://thesafe.house.

Planning

Waterfall Development Process

The game was developed using the waterfall model, which includes 5 principal phases:

1. **Requirements**: discussion of the goals and major features that the game should at least achieve
2. **Design**: research of different technologies and their integration, tentative representation of the game architecture and how it can be modularized
3. **Implementation**: actual game development
4. **Verification**: unit testing, behavioral testing, and debugging
5. **Maintenance**: feature enhancement and further debugging

One big disadvantage of the waterfall model is that the initial requirements and design dictate the development process. The result is the inflexibility of the final software to adjust to new requirements. Instead, we separated requirements that are necessary and feasible from those that are bonus features but require more insight into the actual implementation of the game. As we were developing the game, we frequently came back to the list of requirements and adjusted what could be done and what was no longer essential. Our workflow, therefore, was more dynamic between the five phases and allowed more flexibility in the development process and final expectations.

Technologies

**Collaboration/Communication Tools**

1. **GitHub and Git**

   There are five people in our team, so it was important to collaborate on the same software program without much interdependence. As a distributed version control platform, GitHub allows each of us to have our own local repository, where we could make changes and integrate later with the remote repository. GitHub also keeps track of our work progress and therefore makes it easier to revert unwanted changes and roll back to a specific version of the project. GitHub also lets us create issues/tasks/bugs,
assign them to team members, and monitor their completion. We also used GitHub wikis to publish our weekly logs informing Dr. Vidal of our individual contributions, to establish the goals and requirements of the game, and to make known of our design architecture.

(2) Slack
Slack is a cross-platform communications tool used by teams. It competes with other instant messaging services such as iMessage and GroupMe. In Slack, conversations are grouped into “channels,” and each “channel” represents a specific topic that members can voluntarily join or leave. Our Capstone Computing class utilizes Slack to readily disperse information and provide instant lines of communication between Dr. Vidal, TAs, and students. Joining a channel allows students to get help from communities geared toward specific domains such as Android or iOS. Additionally, we created our own private Slack team to share important updates and set automatic reminders to complete our weekly logs. We also used channels such as “learning” to share documents, notes, and other information that could serve as education tools.

Frontend Frameworks
Five frontend frameworks that we used to implement the game are:
(1) **Angular 2** for building a responsive single page application
(2) **Bootstrap** for facilitating the user interface (UI) design
(3) **Google Maps API** for working with maps and Street Views
(4) **AngularFire 2** for abstracting Angular bindings with Firebase
(5) **HowlerJS** for embedding audio

1) Angular 2

Angular 2 is an open-source web application framework widely used for the development of single-page applications. The following are five notable features of Angular 2:

A. Model-View-Controller (MVC)

Angular 2 adopts the original MVC software design pattern that separates data from its presentation. More specifically, an MVC-based application is broken up into three components: a model, a view, and a controller. For a better explanation of how these three components interact with one another, assume we were trying to create a ship design application where the user can interact with a ship and manipulate its properties such as colors, shapes, and sizes. In this case, a model stores these properties that represent a ship. A view is the HyperText Markup Language (HTML) template that binds to the model to display the ship for the user’s viewing and manipulation. When the user changes any of the ship’s properties through the view’s button clicks or text inputs, these actions are sent to a controller to update the model’s state.

In Angular 2, the functionalities of a view and a controller are combined into a component. A component is coupled with an HTML template that utilizes two-way data binding to enable an instantaneous update of the component’s internal states whenever there is a change in the user interface (UI). In addition, user actions and inputs are sent to the component for interpretation and vice versa, the component can manipulate the Document Object Model (DOM) directly in response to changes in the component’s data. A component can also be composed of different components.
B. Dependency Injection

Besides the MVC design pattern, Angular 2 is also known for its wide use of dependency injection. This technique allows a component to acquire an instance of a dependency while it gets constructed. An example of dependency injection used widely in Angular 2 applications is a service. In some instances, different components might require the same set of operations. To prevent duplicate code, a service specialized in these operations is injected into those components. The use of service injection helps simplify the codebase and makes it easier to maintain.

C. Testing

Another bonus feature of Angular 2 is its integration of testing tools that work seamlessly with a variety of test frameworks such as Jasmine, Karma, and Protractor. Testing is one of the most important steps in software development, and Angular 2 certainly allows us to do so with ease and efficiency with its built-in testing utilities.

D. TypeScript

Finally, working with Angular 2 gives the developers the benefits of using TypeScript over the traditional JavaScript inherent in web development. Even though JavaScript is designed as an object-oriented programming (OOP) language, it is not natively class-based as traditional OOP languages such as C++ or Java, where an object is defined by a class. Additionally, JavaScript does not enforce strong typing and therefore can introduce unexpected type conversion errors at run-time. TypeScript compensates for these shortcomings in JavaScript and encourages safer coding enforcement through classes and static type checking (Feng, 2015). The use of classes allows for code reusability and extensibility through inheritance and extension, making the interaction and relationship between different objects more meaningful. On the other hand, static type checking allows the developers to detect type errors at compile-time before the program runs.

E. Community Support

We did consider React as another potential candidate for our front-end MVC framework. Even though both React and Angular 2 have the same set of features, we opted for the latter because of its rich documentation and a large ecosystem and developer community. These two factors are critical to the facilitation and effectiveness of our future development process.

2) AngularFire 2

AngularFire 2 provides data binding between Angular 2 and our back-end framework, Firebase. Although the Firebase API already offers real-time data storage and synchronization, AngularFire 2 facilitates Angular bindings by associating Firebase references with Angular objects so that they are kept in sync with the remote database (AngularFire). These Firebase references let us subscribe to parts of the database (a single object or an array of objects) and get notified immediately when there are changes to the data. We used AngularFire 2 for better abstraction in client-server communication, thus producing cleaner and more readable code.
3) Bootstrap

Bootstrap is another front-end web framework that we used mainly for the UI. The library contains HTML and Cascading Style Sheets (CSS) templates for common interface components such as buttons, forms, modals, and grid systems. Since none of us are experienced web designers, the use of Bootstrap helps simplify the design of web applications, thus making it faster and more robust. Similar to Angular 2, Bootstrap is very popular among the developer community and, as a result, is more subject to constant bug fixes and feature enhancements.

4) HowlerJS

Besides the UI, we also rely on sound as a tool for storytelling and emotional impacts to enrich the game experience. HowlerJS is a JavaScript library for easily embedding and manipulating audio in a web application. The library supports a variety of audio file formats (mp3, wav, mp4, etc.) as well as full sound control (Howler.JS), thus providing the developers with more flexibility when working with audio in JavaScript.

5) Google Maps API

Since we are developing a Google Street View game, the role of Google Maps API is pronounced. The API provides the application with full access to Street View imagery and location database. We used Google Maps API to embed a Street View and a navigation map in each player’s game view. The API also provides built-in controls for the player to interact with the Street View and the map through navigation, direction, and zooming.

Backend Frameworks

We researched several backend frameworks and platforms that could potentially meet all our needs in terms of storage and reliability. We explored technologies such as Firebase, Django, and a few JavaScript libraries associated with the MEAN Stack. Ultimately, we settled on Firebase due to its versatility and wide suite of services.

1) Firebase

Firebase is an infrastructure platform used to create mobile and web applications. One of the most appealing features of Firebase is its real-time database. We use Firebase to store our starting Street View locations as well as data from each game session. With real-time updating, game data is saved and synchronized between the players and the database. As the user traverses through a Street View, his or her location is updated in the cloud and propagated downward to clients listening to this value.

Firebase offers several other instruments to aid in the development of mobile and web applications. In addition to its real-time database, we could utilize its hosting services to deploy our website. Hosting through Firebase provides us with a fast and secure way to deliver our website to users while employing a Google-backed content-delivery-network (CDN). Although we did not utilize all the features of Firebase such as analytics, authentication, storage, and crash reporting (Firebase: Features).
2) Django

Django is a Python web framework used for rapid web development. It is branded as a fast, secure, and scalable platform. Used by a variety of high-profile companies and organizations such as Instagram and National Geographic, Django is extremely versatile to meet the needs of developers and individuals who interact with it on a daily basis. Django has several built-in functions for templateing, authentication, and security (Django Overview, 2017).

We decided against adopting Django as our back-end web framework. At the time of our research, Django supported real-time updating, but there were concerns about reliability and stability since it was a newly integrated feature.

3) MEAN Stack

MEAN is a full-stack collection of JavaScript libraries and frameworks that aid in the development of web applications. Consisting of MongoDB, Express.js, Angular, and Node.js, these technologies form a toolkit for database, client, and server-side code.

- MongoDB is a NoSQL database that relies on JSON-like documents for storage. It natively supports geospatial information such as latitude and longitude coordinates.
- Express.js is a web framework for Node.js. It provides additional features for web applications and primarily serves as middleware between the web application and Node.js.
- Angular is an open-source web application framework widely used for the development of single-page applications. For more information, please reference Angular 2 under the under Frontend Frameworks.
- Node.js is a JavaScript runtime platform used for server-side code.

After some internal discussions, we decided to move forward with Firebase as our backend framework. Even though we did not adopt the MEAN stack, we were able to incorporate Angular as our web application framework. The MEAN Stack provides the functionality we were seeking, but Firebase is the better lightweight option. Firebase is also easier to implement and integrate with Angular since both are Google-backed technologies (Raj, 2014).

Frontend and Backend Overview

The following diagram (see Figure 1) gives an overview on the interaction between the frontend technologies and the backend platform. We use Angular components to create map and Street View models using the Google Maps API, which are then integrated into the player component. Each component is coupled with an HTML template customized with Bootstrap. The template and the component communicate directly through the event and property binding. We also supply different components with services specialized in certain tasks. For instance, HowlerJS is used to create a sound service that provides some components with audio effects. Services are also used to abstract communication with Firebase from the components. All the data synchronization between the services and Firebase is achieved through AngularFire.
Design

Mockups

We created several mockups to showcase some of the major screens in the app. At the welcome screen, users have the option to select a role or have one randomly assigned to them. Once a role is determined, a role-specific screen provides information about how-to-play. When the game finally starts, each user is presented with a Street View along with a map in the bottom right corner of their page. A list of available items is found on the bottom left of the screen along with the number of items the user currently has in their collection. One of the items that we envisioned was a blur item that temporarily obscures the small Street View that helps the Cat locate the Mouse. When the Cat is near a safe house, a green overlay will appear. Similarly, a red overlay will appear for the Mouse when the Cat is nearby. Once one of the players completes their mission or the time runs out, the game ends.
Welcome Screen

You are the **cat**.
The mouse is nearby. Evacuate to a safehouse.

Your mission is to catch them first.

Share this link with your opponent: [link]

Stage

Cat Intro Screen

You are the **mouse**.
Someone is searching for you. Make it to the safehouse.

Before they catch up to you.

Share this link with your opponent: [link]

Stage

Mouse Intro Screen

Cat’s Game View

Mouse’s Game View
After creating our detailed design mockups, we started outlining how we were going to structure our code. We created a diagram that provides an overview of how components will interact with one another and the backend service. Figure 3 displays the architecture that we initially set out to implement.
We anticipated that when the user goes to our site, the Welcome Component will provide the initial landing page with options to either choose from one of the two roles or have one be randomly assigned. The user also has the options to create an account or log in to view his/her account information and past game matches. After a role is chosen or assigned, the user will be directed to either the “Pursuer” or “Pursued” page for role-specific instructions and goals of the game. The Setup Component will, in turn, generate the game sharing link and initialize the coordinates for the players and safe houses.

Once the users are ready, the Game Manager Component will be dispatched to set up critical functions and services related to the game such as the timer, score, and items. This abstract class also provides the basis for the Pursued/Pursuer Game Manager Components which dictate role-specific UI and functions. At the end of the game, the Results Component will declare a winner and display options to play again or save the results.

**Requirements**

Once we completed the design of the app, we built a list of features and capabilities that we intended to include. They are loosely sorted by importance: core functionalities that we need to complete first, trailed by accessory features, enhancements, and polish. In this sense, the requirements list would also serve as a projection of our timeline.
The first seven requirements were designated as the plan for our working prototype, which was due at the end of the Fall semester. The working prototype was meant to serve as a proof-of-concept that implements at least 10% of the final functionality. For our web app, it also needed to successfully build and deploy on our website to be accessible from anywhere with a browser.

These requirements would serve as a contract between the team and Dr. Vidal by establishing the expectations for our software at the working prototype stage and at final release. As we progressed through the tasks, however, we occasionally made small tweaks to reflect changing needs and priorities. All changes were first approved by Dr. Vidal. The requirements list, finalized on October 13, 2016, is as follows:

a. **Working Prototype**
   1. Optimize for desktop
   2. Deploy on Firebase
   3. Display Google Street View for both players
   4. Display map in corner of Street View
   5. Intercept user input (e.g. keyboard, mouse)
   6. Update location of each player in the database
   7. Show other player's location on your screen

b. **1.0 Release**
   8. Have both players be able to join game via shared links
   9. Displays Street View of the other person
   10. Displays timer on top right/timer works
   11. Items GUI looks good and works as expected
   12. Picked up items are clickable otherwise faded/grayed out
   13. Displays safe houses on map
   14. Detects when pursuer/pursued are on top of each other
   15. Displays result page at end of game with game statistics (Score: time remaining, time elapsed, map of route taken)
   16. Make items randomly appear on map
   17. Implement specific items’ functionality
   18. Blur opponent's screen
   19. Show player location on map
   20. See other items on map
   21. Make the graphical assets (icons, logo, etc.) look good
   22. Displays proximity overlay on Street View for both users
   23. Make the welcome page look good
   24. Create a login page/authentication to add player information to database, allowing them to log in and out. Replaced by: Sound effects.
Implementation

Architecture

The following are the services and components used for the implementation of the game.

A. Services

   Services specialize in certain actions needed by multiple components. Services also have shared properties that allow different components to communicate with one another.

   - **GameManagerService**: oversees the core functionality of the game by initializing data in Firebase and managing elements such as the players, time, and items.
   - **FirebaseService**: facilitates the connection between our app and Firebase.
   - **LocationService**: manages location-related data of players, safe houses, and items, including distance calculations.
   - **ChatService**: interacts with Firebase to send/receive messages between the players.
   - **ItemService**: provides functionality to different item types: blur, blind, burn safehouse, hyperjump, and random hyperjump.

B. Components

   Components are the basic building blocks of the application. They can be reused and combined to create new components.

   - **PlayerComponent**: the parent component for the MouseComponent and the CatComponent, responsible for fetching all the data (player locations, game settings) from Firebase necessary to set up the view and attaching listeners for location and state changes.
   - **MapComponent**: embeds a Google Map with zooming/panning controls and markers to denote the player locations, item locations, and safe house locations.
   - **StreetViewComponent**: embeds a Google Street View with navigation controls
   - **ViewTogglerComponent**: toggles between the opponent’s Street View and the player’s map.
   - **WelcomeComponent**: sets up the welcome page for the game and allows the users to either choose a role to play, let the game randomly choose their roles, or get matched with a random player.
   - **TimerComponent**: creates a countdown timer for the player.
   - **OverlayComponent**: provides an overlay over the Street View to inform the player of the opponent’s proximity. The color and intensity of the overlay depends on the role of the player and the distance between the two players.
   - **ItemManagerComponent**: keeps track of the items collected and used, and performs items’ functionalities.
- **PlayerWidgetComponent**: a wrapper for all the UI-related components besides the Street View and the map (TimerComponent, ItemManagerComponent, OverlayComponent, NotifyComponent, ChatComponent)
- **NotifyComponent**: notifies the player of certain game actions such as item collection
- **ChatComponent**: creates a chat box for the Cat and Mouse to communicate
- **AlertModalComponent**: alerts the player of game disconnections/connections as a result of the other player refreshing/closing the window.
- **LinkModalComponent**: provides the player with a game link to share with the opponent so that they can play against one another.
- **ReadyModalComponent**: asks each player to confirm their ready state and only starts game when both are ready.
- **ResultsModalComponent**: shows the result of the game (the Mouse reaches the safe house, the Cat catches the Mouse, or time runs out).
The UML diagram in *Figure 4* shows a detailed insight into our application architecture by showing how all the components and services interact with one another as well as with Firebase. For simplification, we exclude the components (e.g. modal, overlay, and notify components) that only handle the UI and are not heavily involved in the game logic. There are five types of connections used in the diagram, as indicated in the legend:

1. **Data binding**: used between a service and Firebase to denote a two-way data binding where data is kept in sync between Angular and Firebase.
2. **Dependency**: used between a service and a component to indicate that the component depends on the service for certain utilities.
(3) **Aggregation**: used between two components to show a “has a” relationship. The component at the receiving end of the diamond contains the other component.

(4) **Inheritance**: used between two components to signify inheritance. In the diagram, the CatComponent and MouseComponent both inherit from the PlayerComponent.

(5) **Association**: used between a class and a component (or a service) to show the use of the class and its instances inside the component.

**Challenges/Issues**

We encountered three significant issues during and after the development of the game:

1. **The Lack of a Server**
   
   We mainly used Firebase as our backend framework for its real-time database. Since our game is a multiplayer environment and therefore requires constant data exchange and update between two players, Firebase seems a better candidate than Django and MEAN Stack. Also, Firebase is lightweight and made it easier for us to have a quick jumpstart in our development process. However, there are three limitations of solely relying Firebase without hosting our own server. First, since we do not host Firebase, we have no control over how our data is represented. Firebase stores data in a JSON format and makes the data transfer between the database and our TypeScript objects more transparent (Nathan, 2016). To query a data field, we have to traverse from the root of the database to where that field is nested. However, as our game database grows larger, it becomes harder to query this way.

   Secondly, since Firebase itself is just a database, we could not have any server code. As a result, we handled all the data manipulation on the client side. There is not a clear abstraction between Angular components/services and the database because the former has to communicate directly with the latter. It would have been a better and safer practice if we had not let the frontend meddle with the database directly but only through a server.

   Finally, two players would share information such as time. Right now, the client handles its own timer by using one of JavaScript asynchronous functions, `setInterval()`, to count down the timer every second. Asynchronous events let you move on to another task while waiting for these events to finish executing. Since TypeScript or JavaScript is single threaded, all the asynchronous events are queued waiting for execution (Resig, 2008). Therefore, even when we set the timer to count down every second, this event may take longer than a second depending on the queue traffic. As a result, two different hosts, while starting the timer at the same time, will have different events in their execution queues and therefore will affect the timing consistency. Having a separate server to keep track of the timer and then report the time to the clients will keep the timers in different hosts in sync.

2. **The Drawbacks of Google Street View**
   
   One problem we encountered while working with Google Street View is that not all locations are active for navigation. Some are isolated photospheres that the user cannot leave. In theory, since our game would like to randomly place the player’s initial location every time a new game is started, we could just use the Google Maps API to return a random location. In practice, however, we had to manually preselect these locations to ensure that the player is not accidentally placed in a photosphere. Another drawback of the Street View is that it is not as responsive as our real-time game environment demands. Every time the
player moves through the Street View, he or she has to wait for the Street View to get updated before moving further. This delay slows down the game experience for the players.

(3) Imbalanced Game Logic

During the development process, we failed to devote enough time to ensure the game would be fair for both players. Even though we intended for the Cat to have more difficulty catching the Mouse than the Mouse reaching the safe house, users who tried our finished game suggested that there is not a lot of direction and guidance for the Cat. While the Mouse can see where the safe house is located on the map, the Cat does not know the location of the Mouse and can only guess through the Mouse’s Street View. However, unless the player is familiar with the area where the game is set, it is less likely that he or she could recognize where the Street View is located exactly. For future improvements, we would like to make the game more balanced by at least giving the Cat a direction to where the Mouse is without giving away the exact location.

Validation

Throughout the project, steps were taken to ensure that code worked as intended. These activities can generally be divided into two categories: testing and debugging. Testing involves writing code to automatically detect bugs in a codebase. Debugging involves analyzing found bugs to identify and resolve the underlying problems. Good testing can minimize time spent debugging, because it can point the developer to the source of an issue as soon as it appears. However, we did not experience this luxury in practice; the urgency of deadlines and changing expectations for the architecture led to testing being pushed to the side and frequently delayed throughout the project.

Testing

In this project, two types of testing were employed: unit and integration (also known as behavioral). With a single command (in this project, `ng test` for unit tests and `ng e2e` for integration), these tests can be run to quickly check for errors.

Unit testing focuses on analyzing the code in simple “units,” small chunks of code with clear sets of inputs, outputs, and side effects. Each unit is tested against a representative sample of inputs with known outputs. If possible, these tests cover all cases that require distinct behavior in the function. In practice, it is extremely difficult to cover all the code in a project with unit tests, because units become harder to define on parts of the codebase that interact with external code and data, such as file I/O and API interaction.

Integration testing attempts to cover the entire app at a high level. It applies a different approach to testing than unit testing: rather than analyzing minimal units of the codebase, it tests the codebase as a whole. In our application, we test the app by having Google Chrome, controlled by a program that we wrote, interact directly with the web frontend of our app. This is beneficial because it allows us to test the interactions between code units on a macro level.
Our project did not have 100% code coverage with tests, but in total we wrote 25 unit tests across important components of the app. Due to unforeseen technical limitations with our integration testing tool, only one large integration test was written. We used the following software to accomplish this:

- **Karma**: Automatic test runner integrated with Angular CLI, the interface that we use to manage our project. Runs our unit tests.
- **Jasmine**: Behavior driven test framework used to outline and organize our unit and integration tests.
- **Protractor**: Integration test tool that runs our app in a real browser. We test using Google Chrome. Late in the development process, we became aware of a bug that renders Protractor’s multiple browser functionality incompatible with our app, so we could not run integration tests on the multiplayer parts of the website (Protractor 5.1, 2017).

**Debugging**

Frequently, bugs would appear in the code during development. In addition to Angular and TypeScript reporting compilation and runtime errors through their command line interfaces, we had some tools that supported debugging:

- **Chrome Developer Tools**: A suite of tools that allow developers to debug code, tweak HTML/CSS, and study networking between the browser and server. Because we used TypeScript, a language that compiles to JavaScript to run in the browser, we modified our configuration files to generate source map files, which map between the compiled JavaScript and the TypeScript source files. These files allow us to debug the TypeScript files directly.
- **Travis CI**: Continuous Integration tool. Whenever a commit is made to our master branch, Travis pulls the code and performs a clean build.

**Results**

**Finished App**

Our finished app is the culmination of two semesters filled with collaboration and teamwork. Deviating from our design mockups, we made significant changes to improve the views and clarify the goals and objectives. When users visit our site, they are now greeted with one of the preset Street View locations of a scenic area. Scrolling down, users can learn more about *safehouse* and get information about how-to-play. Similar to the mockups, the Cat and Mouse views display a Street View with a small map in the bottom right corner. Both views remain mostly identical except for role-specific items and a small Street View in the Cat view that displays the Mouse's location. We also added more items such as *hyperjump* that allows users to "jump" to preselected locations throughout the city. As players get closer to one another, a color overlay and notification will appear indicating imminent success or failure. Once
the time runs out or one of the players completes the mission, the game will conclude with a results screen. Below are several screenshots of our app.
Presentations

To close out the semester, we presented our final app on April 21, 2017 to Dr. Vidal and our peers in the Capstone Computing class. We were also invited to present a poster to the Industrial Advisory Board (IAB) on April 28, 2017 where we met with board members and CEC faculty.
References


GeoGuessr - Let’s explore the world!. (n.d.). Retrieved from https://www.geoguessr.com/


Google Street View: Where We’ve Been & Where We’re Headed Next. (n.d.). Retrieved from https://www.google.com/streetview/understand/
Appendix: Tools and Technologies

Here is a list of some of the major tools and technologies we used in this project, along with links and short descriptions of their purposes.

**Angular 2:** Development platform used to build the client side of our web application.
https://angular.io/

**Angular CLI:** Command line program to build and test our web application.
https://cli.angular.io/

**AngularFire 2:** Official Angular library for Firebase interaction.
https://github.com/angular/angularfire2

**Balsamiq:** Wireframing tool used to develop the design mockup for the app
https://balsamiq.com/

**Draw.io:** Flowchart and Diagram Software, used to create the architecture document and Figure 1: Front-end and Back-end Interaction.
https://www.draw.io/

**Firebase:** Web application platform used to host our content and implement a database that synchronizes both players in a game.
https://firebase.google.com/features/

**Gimp:** The “GNU Image Manipulator Program.” Used to make and tweak graphical assets in the game.
https://www.gimp.org/

**Git:** Version control system used to handle the project.
https://git-scm.com/

**GitHub:** Collaboration platform and host for git repositories used to coordinate the team throughout development and planning.
https://github.com/

**Google Chrome:** Web browser supported by our web app. Also used frequently during debugging for its rich array of developer tools.
https://www.google.com/chrome/browser/index.html

**Google Maps API:** Library used to display the map and Street Views on screen, handle user input, and handle data sent to/received from the server.
Karma: Automatic test runner integrated with Angular CLI, the interface that we use to manage our project. Runs our unit tests.
https://karma-runner.github.io/0.13/index.html

Protractor: Integration test tool that automatically interacts with our app using Google Chrome.
http://www.protractortest.org/

Sublime Text 3: Text editor used by some team members in development.
http://www.sublimetext.com/

Travis CI: Continuous Integration tool. Whenever a commit is made to our master branch, Travis pulls the code and performs a clean build.
https://travis-ci.org/

VS Code: Text editor used by some team members in development.
https://code.visualstudio.com/

WebStorm: IDE used by some team members in development.
https://www.jetbrains.com/webstorm/

Jasmine: Behavior driven test framework used to outline and organize our unit and integration tests.
https://jasmine.github.io/