

# mmOOG : A Multiplayer Online Google Maps Trading Game

Emma Persky `esp05@doc.ic.ac.uk`

June 4, 2007

## Abstract

In this report I will explain the methods and principles used to create a multiplayer, almost-realtime, trading game, playable in any modern web browser. The game will use Ruby on Rails as the web framework, Google Maps API as the core of the game board, and Javascript to run the game client.

## 1 Introduction

The overall purpose of this project is to ensure acquaintance with current web programming, web / database integration, and the client / server model. The specification detailed a space trading game, but left scope for redefining this, as long as a few simple requirements were met.

- The game must be multi user, and played through a web browser.
- The game must implement a database to save and restore from game state.
- The game must involve some form of inter-user communication, e.g. trading and messaging.

## 2 Game Structure

The game is designed to be an ongoing multiplayer game. There is no specific start or finish, although the game can be restarted at any point, and a winner could be declared. It's rules have been designed such that it would be possible for a new gamer to have a chance of overthrowing a long standing incumbent game leader. It is based around exploring a virtual map of the real world, and is a finance based game in the sense that the leader is the player with the most money in his or her bank. The game is realtime (allowing for a few seconds for information to propagate over the network, and be updated to other players) and as such there are no specific turns - the longer you spend logged into a game trying to earn money, the more you can earn!

### 2.1 Gameplay

The game is initially started with a set of cities (in their real world locations on the map), and a set of transport routes (roads), these are somewhat arbitrary, and should be changed if the game is restarted so that players are not able to predict game development. More cities leads to an overall longer game before one player is clearly in the lead, and more connections between cities counter's this by making it hard for one individual to claim a clear lead. Cities can contain a train station and an airport, which can be used by players to create custom transport routes, which can cover long distances (and so initial transport routes should only cover short distances).

### Gameplay Rules

- New players are inserted into the game at a random city.
- Players move between cities by traveling along transport routes, but they must pay a fee for each city they travel through. This fee is allocated as specified below.

- Players may only move one city at a time. After arrival at a new city, they may select their next destination.
- Players can earn money by completing missions, which are to reach given cities within a given time frame. Missions span real time, so a player can not log out and in to change their mission.
- Players may purchase, disband, or trade cities, train stations, or airports.
- Players may create transport routes between any city with a station or an airport (i.e. they can not create roads), but they only have to own one of the facilities. These routes have a faster transit speed than the initial routes, but cost to travel along. Only one transport route may exist between any two cities.
- To trade, purchase or disband a city or facility, the player initiating the action must be at it's location.
- Disasters can happen to cities, facilities or transport routes, rendering them unusable. They can then be rebuilt by anyone.

## Game finances

- Each player has a global bank account which starts with a balance of £5000.
- An unowned city costs £1000 to purchase.
- An unowned station costs £500 to purchase, and an unowned airport costs £800.
- City facilities can only be purchased in owned cities.
- The fee for traveling through a city depends on how many connections that city has. The fee is £50 per connection to a maximum of 10 connections, and is only taken if the city is owned. It is distributed as 60% to city owner 20% to each facility owner, or back to the central pot if either does not exist.
- The fee for traveling along a private transport route depends on it's length, and is specified as £1 per 25 miles on a train and £1 per 10 miles for airplane, and is distributed 60% to route owner and 20% to each end point owner.
- The fee for building a transport route is 10 times that of traveling along one. It is possible to build airplane routes to cities without airports (to initialize overseas travel), but the cost is twice that of building a standard transport route.

## 3 Game Architecture

As with most multiplayer games, and as specified in the requirements, the game employs a client / server architecture. The game client runs in a web browser, and should talk to a web server (that served the web page) for game updates. This communication should all be over HTTP to avoid any firewall interference.

### 3.1 Server Architecture

The server is a web server running Ruby on Rails and a MySQL database. There are many ways to run Ruby on Rails, and it is becoming a standard for Web 2.0 applications. Current trends [1] indicate running the Mongrel Web server (which actually process requests to a rails application) in a cluster formation behind a proxy balancer running on apache. The balancer spreads the load between each Mongrel instance which is only single threaded so can block if there is a long request. Each Mongrel instance talks to the same MySQL Server instance, and as such it is not important which instance any given request runs on. Whilst this may seem somewhat out of the ordinary compared to running a php application, it allows something about ruby is better.

Because of the nature of Ruby on Rails, the game can be redeployed on any machine that is capable of processing a CGI request, and running the ruby script parser. In addition it is possible to run the back-end database on almost any server, as Rails is able to talk to many different databases, without the need for extensive reconfiguration.

## 3.2 Client Architecture

A few years ago, web based systems were clunky and hard to use. Simply changing one page option or requesting some form of update required a full page refresh and redraw. This was slow, ugly and not a pleasant user experience compared with installed applications. However, modern web design techniques allow developers to create websites whose look and feel is much more friendly, and can avoid these ugly page refreshes. These modern techniques are not suitable for all websites, as it can be reassuring to see the page refresh in some actions, however I felt that this is not synonymous with good game play. For a game to feel playable it must avoid sudden actions and complete redraws, actions should be smooth and flowing and the player should not have to wait between actions (except where the rules define otherwise).

The three core ways that such interfaces can be created on the client are Flash, Applets, or lots of CSS and Javascript. I felt that in order to reach the widest possible audience the game should avoid the need for any additional software installations, and whilst Flash and Java are included on many systems, not all have them installed by default. By contrast the use of CSS and Javascript is native to all fully featured modern browsers, and as such the choice was obvious. Using Javascript, all communications can be made asynchronous and responses can be parsed rendered to the page without the need for a refresh.

## 4 Communications Structure

Client / Server communication is divided into three distinct areas, initial data download, periodic asynchronous updates, individual quasi asynchronous actions. The former two use strict AJAX principles calling a server resource and then processing an XML result. The latter does not expect an XML response, as the result is usually a simple id or failure message. I had a choice between JSON (Javascript Object Notation) and XML (eXtensible Markup Language) for the return value, each has their own specific benefits and pitfalls. XML is more readable, but JSON is more lightweight, XML is more of a standard, JSON requires less client side work because it is native javascript. Ultimately I chose XML because of it's tight integration with Rails. Rails exposes functionality to express the structure of an XML document to be created from a dataset, in the form of a .rxml document, which proved to be very simple to use.

### Statefull Communication

Because session state is maintained between the client and the server, it is possible to determine which players has called which action by storing the logged in player's id in a session variable. As such is it not necessary to pass information to the server about which player is called which action. This is particularly important in the individual updates aspect.

### 4.1 Initial Download

The initial data download is instantiated when the game initially loads, and can be, depending on game complexity, a fairly significant download. For example, a game with 500 cities, 50 players and 1000 transport routes would have an initial data volume of around 3 megabytes, which even on a fast internet connection could take a minute to download. The 5 initial download sets (the additional two are active trades and messages, but these are insignificant to the total data volume) are requested and processed simultaneously to speed the process up. The server passes back an XML document for each request, which are parsed, and the data they contain instantiated as objects and stored in a set of hashes for easy access.

### 4.2 Periodic Updates

Once the game has been loaded, and game play commences, the game instantiates a periodic call to /player/update asynchronously every 2 seconds by default. This figure is adjusted using the network connection setting, so that users on faster networks can play a more real time game, and those on slower networks do not choke their connection. This call represents the bulk of the inbound communications, and includes all notifications of messages, trades, player moves, new players, etc, etc. A sample document is shown below. The document contains two types of elements, updates and moves. The idea behind this is that efficiency can be fine tuned so that not all moves need be distributed to all clients, and a more complicated selection procedure is needed to work out which moves to display, and as such a different set of xml needs

to be generated. The update controller on the client parses the document and takes the appropriate actions, such as moving other players pieces, flashing notices of trades, etc.

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
  <updates>
    <update update_type="bank" detailt="add:53" />
  </updates>
  <moves>
    <move player_id="1" start_id="2" end_id="7" />
  </moves>
</document>
```

### 4.3 Player Updates

The final communication element in the game is that of informing the server of updates made by the player, such as moves, purchases, trade requests, emails, etc. These happen in two ways, either as a simple call to a server resource with an ID parameter, or as a more complicated post body call. In these, a hash is produced of the data that needs to be passed to the server, and attached as the post body to the AJAX call. These functions all observe the response from the server to ensure that the action is valid. Invalid responses generate a game error, and require the user to log out and log in again. These are only expected if the user is attempting to cheat at the game, or if there is some form of data corruption.

## 5 Client

The game uses a combination of HTML, CSS and Javascript to create a playable environment. The HTML is used purely to lay the document out in terms of content, and contains no information about the styling of those elements, which is completely contained within the CSS in the style sheets. Not only is this good practice in terms of creating a clean boundary between content and layout, but also means that cross browser compatibility issues can be resolved by delivering the appropriate style sheet for the supplied USER AGENT header. Javascript is used to animate and respond to actions in the user interface. The game interface can be considered in three parts.

- Registration and login.
- The user controls.
- The game board.

### 5.1 Registration and Login

Registration is carried out outside of the core game environment, as there is no need to load all of the game data and javascripts if the user chooses not to register or login, and only want to view the home page, which contains information about the game. A player may enter their credentials on the home page, and click login, at which point they are directed to the game itself if they are validated, or informed of the error. New users may also register on the homepage, and are taken directly to the game after a successful registration. New registrations require 'human verification' which entails entering some text into a field which is displayed in such a manner that it would be unlikely that a computer would be able to decipher (e.g. with random lines superimposed on the text), to ensure that 'spam bots' do not create fake registrations, which can often overload a system.

### 5.2 The User Controls

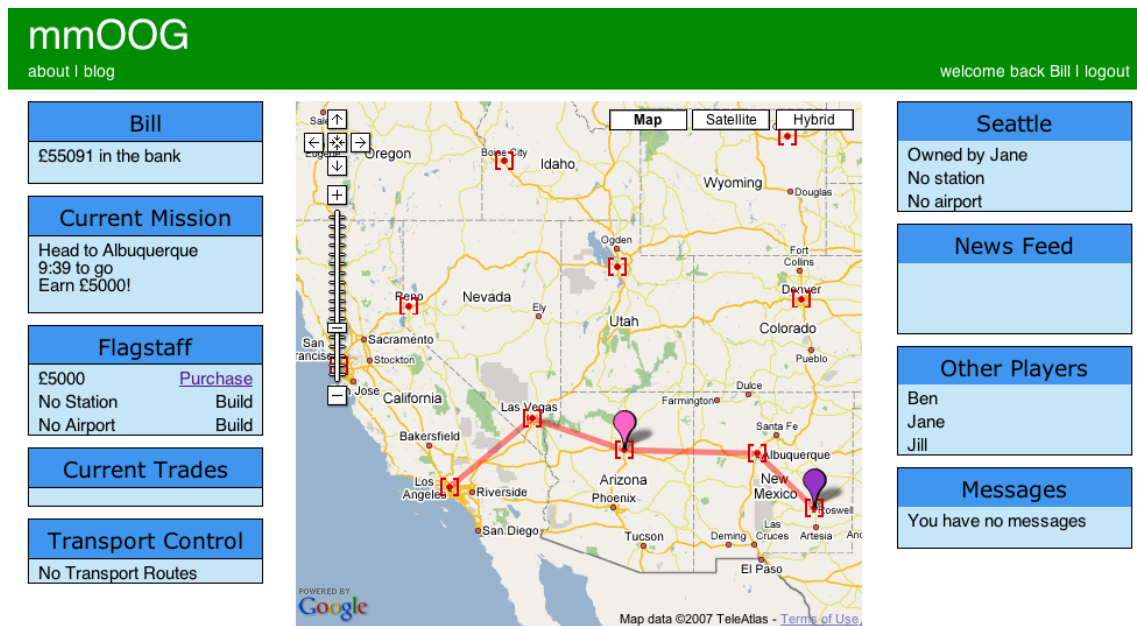
The user controls are laid out to the left and right of the game board. Those on the left represent actions directly relating to the player and their game play, and those on the right are more general. The control boxes can be minimized if the information is not of interest to the player. If, for example, a player is not interested in any trades, they can minimize the trade information box. The Newsbox is updated with

information about other players, for example, if other players complete a trade or purchase. The Tradebox is updated with information about trades pertaining to the player, likewise with the Message box, and so on.

### 5.3 The Game Board

I felt that the best choice for a game board would be a Google Maps API map [2]. Many players will already be familiar with using Google Maps, and if they are not, it is designed to be easy to pickup. The infinitely scrollable world surface lends itself well to games played out across real world locations, and the API provides excellent facilities for overlaying custom markers, icons, lines, etc. Google have designed their Map to be cross browser compatible, which takes away a significant proportion of the headache of dealing with multiple browsers on multiple operating systems.

### Screenshot of mmOOG



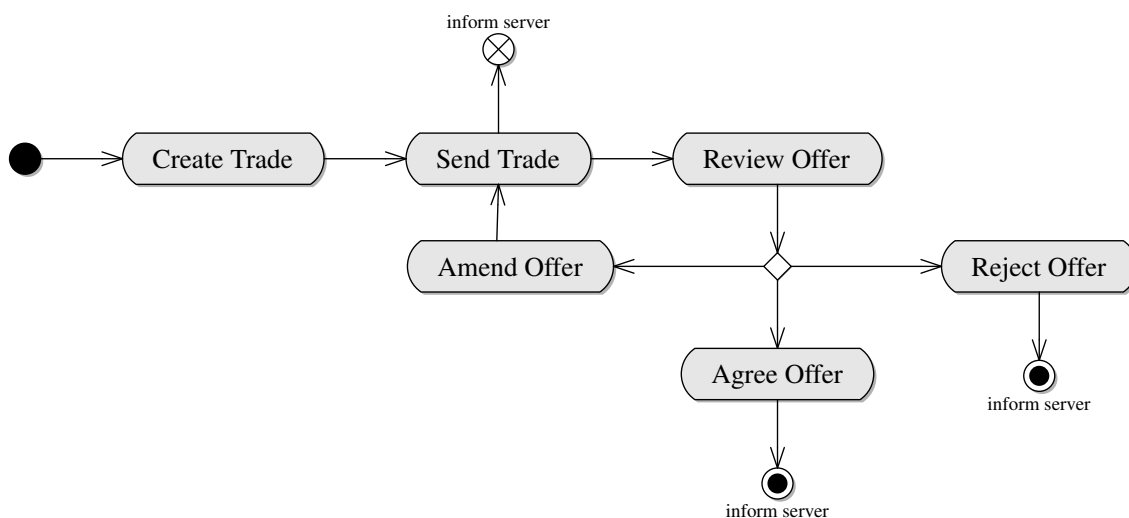
### 5.4 Trading

Trading can take place between two players when the trader (the player who initializes the trade) is at the city which forms part of the trade (city, station, or airport). Once the trade has been initialized, the player may move on, and continue. Offers can go back and forth between the trader and the tradee, along with messages. If a player makes an offer, it is binding, the other player may accept the offer and the trade goes ahead. Any player may reject or cancel their offer at any time, at which point the trade finishes. Once a trade is complete and agreed, the money for the trade is subtracted from the trader, and held until the players must meet (any where on the board) at which point the transfer happens. Trading happens on a screen that appears on top of the main game control, although the game continues beneath. New trades and trade updates are notified in the Trades section of the control panel.

### 5.5 Messaging

Messaging works in much the same way that trades do, infact the display code is the same, except the trading controls in the screen are hidden. New messages are notified in the Message control panel to distinguish them from trades, which may be of more, or less, interest to players.

## Trading Diagram



## 5.6 APIs

In addition to the Google Maps libraries, the project use two further freely available libraries. The script.aculo.us Effects Library [3], and the Prototype Javascript Framework [4].

### 5.6.1 Google Maps API

The Google Maps API is provided as a javascript include that downloads directly from Google's servers. It provides its functionality by allowing access to a set of custom Javascript Objects. For example to included a map on the page a new GMap2 object is created and passed a reference to the <DIV> element where the map should be located : `map = new GMap2 (map)`

### 5.6.2 script.aculo.us

script.aculo.us is used for most of the visual eye candy that is not directly related to items on the map. It has been developed to ensure cross browser compatibility and so can be used without worrying about which browser the player is using. It provides features such as `Effect.Fade()` and `Effect.Appear()` that can be used to fade items in or out of the display. The library contains a set of core features () which can be extended and combined to produce a myriad of effects, that until recently were very difficult to display in a web browser. script.aculo.us (along with Prototype) is distributed as part of the core Ruby on Rails distribution.

### 5.6.3 Prototype Javascript Framework

The Prototype Framework extends the core javascript implementation that is included in every web modern web browser. The 4 main features used within this game is functionality like DOM Object selection, javascript class creation, fully enumerable hashes and arrays, and cross browser AJAX compatibility. DOM Object Selection `$(obj_id)` function returns a reference to the DOM Object whose id it is passed. This circumvents the need for a lengthy piece of code to ensure cross browser compatibility. Javascript class creation is crucial to the operation of the game, as each game element is represented as a javascript class. Prototype exposes the `Class.Create()` method which return a reference to a class to which functions can then be attached. These classes can then be instantiated by javascript in much the same way that they can be in any object orientated programming .

## 6 Server

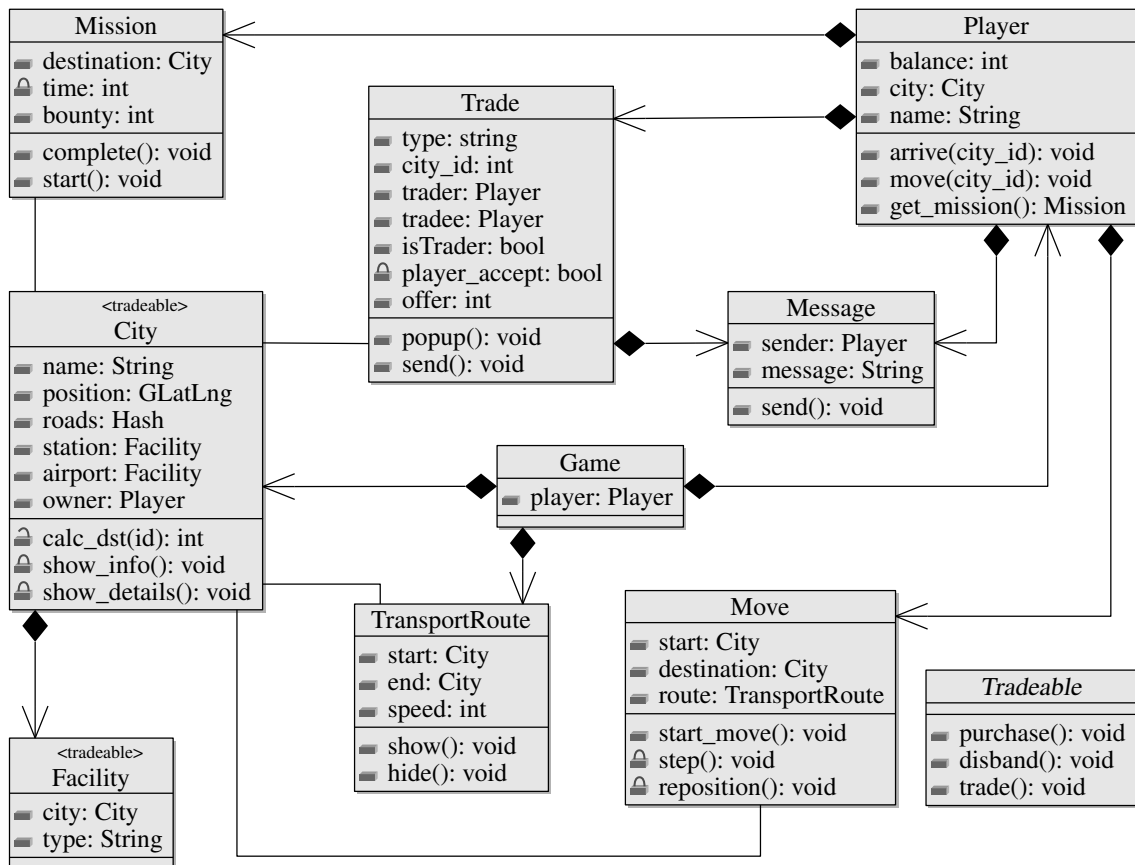
I chose to write the server aspect of the game in Ruby using the Ruby on Rails framework. I recently decided to learn the Ruby programming language for the purpose of getting to grips with development in

Rails. Rails is based around the concept Convention: if there is a conventional way of doing something, that is the way Rails assumes it is done unless you specify otherwise. In this way, it is able to automatically create object / relational mapping, by assuming that field names in the database map directly in type and name onto members of a class. Of course there are times that is necessary to override the default mapping, or to create custom database handlers, but for the majority of fields this is not the case. Rails also manages what is known as database migrations, whereby a set of files denote what fields and tables to add (and remove) from the database. This allows the database (and thus the Classes) to be rolled back to a specific version if needed.

## 7 Objects

A set of objects representing game elements are written both in Ruby for the server, and Javascript from the client. A UML diagram of the Javascript objects can be seen below. On the client, objects themselves are responsible for interfacing with the Google Maps API through a single function. This ensures that if the Google Maps API changes, or I wish to change the game board at any point, it is a simple process to update the relevant functions.

### Game Class Structure on the Client



### 7.1 City

The City class exists on both the server in Ruby and on the client in Javascript. It maintains a list (in a hash on the client, in the database on the server) of TransportRoutes that are associated with it, and stores references to its Facility objects. It implements the Tradeable interface so that it may be traded. The city position is sorted in a Latitude and Longitude pair. In the client this is stored in a GLatLng object for use by the Google Maps API. In addition it contains a piece of code to calculate the

distance between itself and any specified city in the method `calculate_distance(city_id)`. This method approximates Earth as a perfect sphere, and calculates the distance across its surface [5].

## 7.2 Facility

The `Facility` Class is used to represent city facilities, specifically Train Stations and Airports (stored in the `type` member). This was created to be inherited from, if necessary, to create other city facilities, but the current set up does not require this.

## 7.3 Game

The `Game` class exists only in javascript, and manages all aspects of initialization of game data and creation of game new objects instructed by the server(e.g. a new trade), and crucially it manages the periodic updates. It also contains functions for purchasing, disbanding, trading and messaging (e.g. `city.create_trade()` which creates a new trade object. The `Game` class also contains hashes that hold references to all of the game objects that can be considered in collections (`Player`, `TransportRoute`, `City`, e.t.c.), indexed by the object ID that is in the database.

## 7.4 Message

`Message` objects are created for messaging in the game, a new message is stored for each message created, or they can be used in a conversation basis where the whole conversation is stored in the object.

## 7.5 Mission

The `Mission` class is used on the client to store the mission details include destination city and allowed time, and to operate the timeout mechanism that counts down until the time to complete this mission is over. This is implemented with a recursive call to a function `step()` which checks if the time has expired, and if not recalls itself after 1 second. On the server this class contains the logic for creating a new mission. When a new mission is created all that it needs to know is the `Player` to which the mission is being assigned, it then randomly selects a city from the database, and ensures that the the city chosen is not the same as the previously allocated mission. The `complete_mission()` function on both the client and the server updates the bank. On the client this is called from `Player.arrive()` and on the server from a service of the same name.

## 7.6 Move

A `Move` object is instantiated everytime a player's piece must be moved on the board. It takes into account the users preference for computer speed to determine the frequency and size of move steps. This information is then used to remove and redraw a players piece on the board in a different place along the route until it reaches its destination city. `start()` set's up the calculation and starts the recursive call to `step()` which calls `reposition()` which actually moves the player's piece, and sets up a call to itself after the calculated interval.

## 7.7 Player

The `Player` class is used to store information about Players (note: not specifically the current player). It is used primarily on the client for creating `Move` objects for other players, and to store their name, stats, etc. The specific instance of the class relating to the current player is used for informing the server about updates to the player, for example, `Player.arrive()` is called by `Move` when a player's piece arrives at a city. On the server the class contains logic to ensure that the player does not arrive (or leave) a city they should not be at. It also contains code for acquiring the Player's mission. This code ensures that if a player has already requested a mission but, for example, loses connection and rejoins the game, that they are allocated the same one if it's time has not expired.

## 7.8 Trade

The `Trade` class manages trades. It stores references to the trader and tradee, what is being traded, the current offer on the table, and a reference to a message object that stores messages associated with the trade. The code ensures that if an offer is made to a player, they may accept that offer, reject it outright, or place a counter offer at which point they may no longer accept the original offer. The code on the server strictly mimics the behavior on the client to ensure that trades can not happen outside of the trading rules.

## 7.9 TransportRoute

The `TransportRoute` class is responsible for storing information about routes that connect cities. It contains references to the player that created the route (if it is not a basic road), the type of route and the cost to travel which it caches when created to avoid calls to the function to determine the length of the route which is mathematically quite heavy. On the client it also contains functions `show()` and `hide()` which allow the road to be hidden from view by the game controller.

## 7.10 <<tradable>>

The <<tradable>> interface exposes the methods required to make an object tradable, specifically `purchase()`, `disband()` and `trade()`. `purchase()` checks that the item is available, removes the funds, and updates the server so that other game clients may be informed. `disband()` is similar except it reallocates funds. `trade()` is somewhat different in that it sets up a new `trade` object with the default parameters, such that the trading screen may be displayed.

# 8 Conclusion

This has been a highly enjoyable project, and having the chance to create one of the very first multiplayer games based on google maps has been a great driving force for me. Whilst many of the technologies I used in the creation of the game were not new to me, I feel that I have been able to hone my skills, and improve my technique.

## 8.1 Working Alone

It was my decision to work alone in this project, and I feel that this was a wise choice in this instance. Whilst at times it would have been beneficial to bounce ideas off of a group, and have the combined wisdom to help see problems before they occur, I do not feel I have lost out. I believe the size of this project was on the boundaries of what can be achievable by a single person. If I was in a group, I have no doubt that many of the features I would like to implement and had ideas for may have had a chance at being developed, but as an individual my time was far more constrained.

## 8.2 Language Choices

Javascript and Ruby on Rails were, I believe the right choices for this project. The javascript libraries that I used allowed for very fast development of client code, and simple modification where needed. It is fully cross browser compatible (because of the inherent nature of the chosen libraries). I now have an additional project of Rails development in my portfolio, and grow happier with the language and framework everyday. I firmly believe that Rails is going to be one of the leading choices for web development over the next few years, and this is already becoming evident from the large number of sites that use it [6].

## 8.3 Future Development

After the final deadline for the project I plan to review my work and release it to the public live on the internet. I am interested to see how a server can scale to handle the calls to it, and how many simultaneous users the Google Maps API can handle. I would be interested to see if I have unintentionally created any network bottle necks, and attempt to resolve these by perhaps defining a more terse communications schema. Assuming that the server process is able to handle many simultaneous users, I believe the problem

might be the number of players being displayed on the map. Each player requires a significant amount of memory to be drawn, and even more to be moved. If this does prove to be a problem my solution would be to write a function determining if any given Latitude / Longitude pair lies within (or just around) the Map viewport, and only then draw icons satisfying this condition. The downside of this is that there would be a delay when rapidly scrolling around the map, and that I would have to restrict the zoom out levels to prevent the entire world being displayed. All of these are interesting challenges that I look forward to solving.

## 8.4 Future Game Rules

As for game rules, I would like to increase the city facilities to perhaps include schools, hospitals, local transport, etc, which could be used to create gameplay whereby users must manage individual city infrastructure, and could earn credit for well managed cities. In addition cities could be linked in some way so that management could be centralized. Some form of AI could be introduced to determine how well a player is managing their cities.

## Learning Outcomes

- I now know how to use  $\LaTeX$
- I have a good understanding of MetaUML[7]
- Google Maps API experience
- Subversion for Code management
- TextMate [8] for Development

## References

- [1] Time For A Grown-Up Server: Rails, Mongrel, Apache, Capistrano and You, Coda Hale, Internet, <http://blog.codahale.com/2006/06/19/time-for-a-grown-up-server-rails-mongrel-apache-capistrano-and-you/>.
- [2] Google Maps AIP, Google, Internet, <http://www.google.com/apis/maps/>.
- [3] script.aculo.us, Thomas Fuchs, Internet, <http://script.aculo.us/>.
- [4] Prototype Javascript Framework, Sam Stephenson, Internet, <http://www.prototypejs.org/>.
- [5] Latitude and Longitude formula, Chris Michels, Internet, <http://jan.ucc.nau.edu/~cvm/latlongdist.html>.
- [6] RealWorldUsage in Ruby On Rails, Rails Website, Internet, <http://wiki.rubyonrails.org/rails/pages/RealWorldUsage>.
- [7] MetaUML, UML Diagrams for  $\LaTeX$ , Internet, <http://metauml.sourceforge.net/>.
- [8] The Missing Editor for Mac OS X, Internet, <http://macromates.com/>.