Gunship Galaxy

Introduction

When we first got together, there were obviously a lot of ideas thrown around the table. Building a WebApp is such a broad specification that it was hard to know where to start. Our first big decision was platform: where did we want people to use our app? We first met the morning after Google I/O 2013, and the demonstration of the new app facilities there (several new APIs, the Android specific IDE based on IntelliJ, to name a few) had a couple of us rooting for Android development. We put that idea to one side, however, as we realised that basing our decision to develop on Android on a flashy presentation was probably unwise.

At this stage we took stock of the devices we used individually. Two out of the four of us had iPhones, and the other two had Android phones. It was then that something else from Google I/O inspired us. We all thought Chrome Racer was pretty cool. So much so, in fact, that it was at least an hour before we did anything productive after we found it. We liked the idea of using Chrome as a kind of ubiquitous platform across devices, and started thinking about how we could do something interesting with it.

After a few iterations, we came up with the idea of using one device as a “monitor” and a number of other devices as “controllers”. We then discussed the potential issue of latency. Firstly, we could design a game where latency would not be an issue in the first place (such as something like Scrabble, where each player could hold their letters on their device, and the “monitor” could show the whole board). However, we realised that this had the potential to be fairly unimpressive.

At the other end of the spectrum, we could design a fast-paced game where latency had the potential to be problematic, but which would be much more impressive when it worked. We decided that the game we ended up with need not be too complex, as successful internet games often are not (see QWOP, or Boxhead), and that we could find a sweet spot with respect to the latency issue and the pace of the game. Websockets seemed to guarantee a certain degree of speed in connections, and this was a reassurance.

As such our goals were as follows:

1. Using Websockets, design a way of controlling a “monitor” session with a “controller” device.
2. Design a simple game which could be played with it.
3. (Theoretically:) Have multiple potential games to play using this system.

Figure 2 – Gunship Galaxy opening menu, unpolished

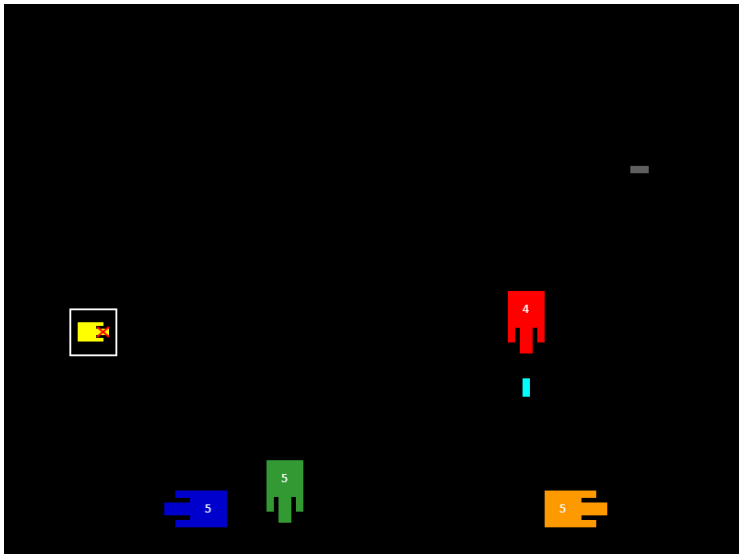


Figure 1 Gameplay screenshot

Overview

Our WebApp is a multiplayer 2D arcade style chaotic shooter. Its main appeal – despite the obviously brilliant gameplay – is its retro style, and the ability to play across multiple devices with friends. A basic game session consists of:

- Logging in
- Setting up your game
- Battling against your friends or versus AI.
- Repeat last 2 steps ad infinitum.

Gameplay

The concept is simple, you pilot a gunship, and the aim is to destroy all the other ships by shooting them enough times. The last one standing wins.

Your ship goes forward all the time, and there is no way to stop it. All you can do is rotate the ship to go left/up/right/down – the only 4 possible orientations – and shoot in front of you. If you reach the edge, your ship automatically turns 180°.

Bullets travel at twice the speed of ships. When they reach the edge, they bounce back and their speed doubles. If they reach the edge again, they are destroyed. If a ship is hit by a bullet, it loses a life. If a ship has no life, it is destroyed.

Bonuses enrich the game, and are what makes it so replayable. Bonuses can randomly spawn and, then be picked up. Examples of bonuses include:

- Invincibility: For 4 seconds, if you are hit by a bullet, you do not lose a life
- Gun disabling: For 4 seconds, all enemy ships have their gun disabled
- Life: Gain one life
- And many more...

When a player wins the game, the data of the game is sent to the database – final rankings, how many times each ship has been hit/has hit somebody. This data makes up the users profile, and in the future, could allow other players to compare stats, and have leaderboards.

Design

From a design perspective, the project is separated into two main parts:

- The game logic and display
- The database and servers

Game logic and display

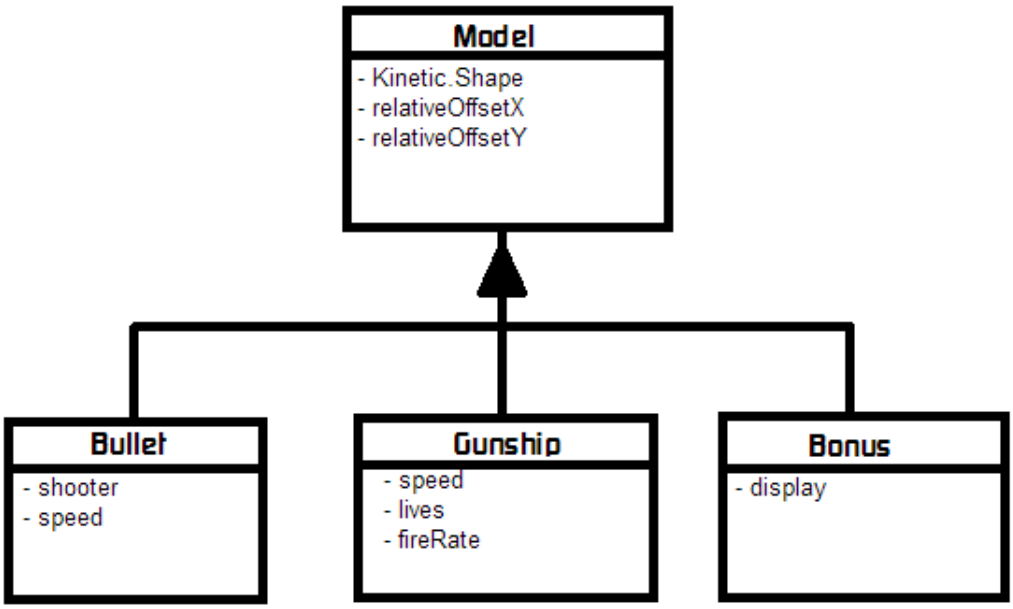
The game works from a “main” class, gunshipgalaxy.js. This class initialises the main menu, and then switches to the class menu.js to handle everything from the menu. Once all the settings for the game have been put and the button “Play” has been pressed, menu.js switches back to gunshipgalaxy.js that creates all the gunships and the main animation to update the positions.

All the visible objects (gunships, bullets and bonuses) sort of inherit the same class, in the sense that they all have a field “model” which stores information about the shape, position, rotation of them. This allowed us to use a lot of template methods, even if JavaScript does not have a notion of classical inheritance: methods for advancing, detecting collisions, destroying objects are based on the model and are therefore common.

Bonuses all have the same model, even if they do not have the same display, nor do they have the same effect when picked up. This also allowed us to have a constructor for that and prevent duplications or errors due to a different implementation of the bonus model.

Since JavaScript has no concept of field visibility, we did not need any controller class that handles the logic; every object works on its own: gunships shoot bullets directly, bullets are created according to the gunship coordinates which are accessible, they also explode by themselves and call a method in gunship to express the fact that they exploded on a given gunship. There is no controller class implemented to do that.

Finally, we have a few listeners: we decided to use the Kinetic.Animation class to work as an observer for its simplicity to use and conciseness. For example, bonuses are attached an animation when created, so that they are displayed and checked on to know when a gunship catches it, and call the appropriate method in such a case. The user inputs are dealt with the JavaScript common feature of events, to change the options of the different spaceships in the menu, change the orientation of the gunship in the game, shoot a bullet...



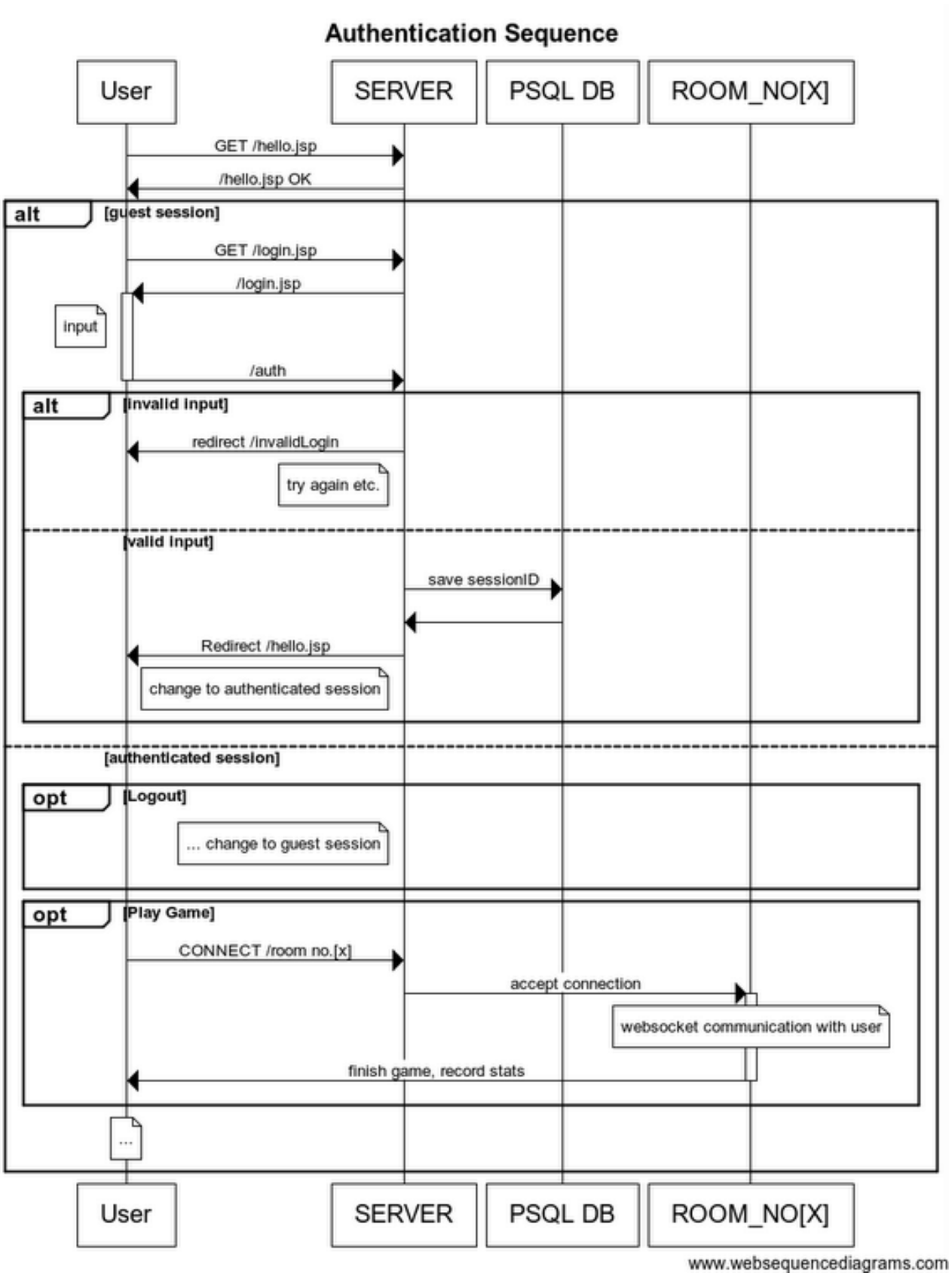
Design process – server side

The original aim on server-side was to make the interactions between the players of the same session as close to real-time as possible. Standard HTTP or AJAX polling could not fulfil this demand as they would have required significant communication and/or process creation overhead. That is the main reason why we gave up on classic Apache and (Python-written) CGI as the original server-scripting language pair we had considered.

Instead, we resorted to an Apache Tomcat – JSP architecture, since it provides the possibility for real-time communication in the form of WebSockets[1]. Furthermore, the servlet system provided higher flexibility in terms of access control and session settings (such as deploying a 30-minutes-since-last-access session for logged-in users.)

Below is a sparse sequence diagram describing how a session would work in general. It doesn't capture all the possibilities: a guest user can still play the game with an anonymous ID, but the stats won't be recorded in the database. Also, there are attempts to filter down unfair access to the server by setting certain session cookies or by specific tomcat mechanics (so, for example, if you try to logout as a guest, you will be redirected to an explanation page saying you can't logout as a guest. There is also a protected folder of things which can't be accessed. Etc.)

Once connected, the players can agree to start the game. Once the game is finished, the stats are recorded to the database.


Authentication Sequence
User
SERVER
PSQL DB
ROOM_NO[X]
GET /hello.jsp
/hello.jsp OK
alt
[guest session]
GET /login.jsp
/login.jsp
input
/auth
alt
[invalid input]
redirect /invalidLogin
try again etc.
[valid input]
save sessionID
Redirect /hello.jsp
change to authenticated session
[authenticated session]
opt
[Logout]
... change to guest session
opt
[Play Game]
CONNECT /room no.[x]
accept connection
websocket communication with user
finish game, record stats
...
User
SERVER
PSQL DB
ROOM_NO[X]
www.websequencediagrams.com

Group Structure

- Mihai: server side communication.
- Will: client side login, backup/version control.
- Conrad: front end game development, sound design and team coordinator.
- Guillaume: front end game development, graphics.

Group organisation

We met twice a week on Mondays and Thursdays in order to ensure that we were all on track. While at these meetings, we would go through current features, features we'd decided not to continue working on, and features that need fixing/had been fixed. Here is a sample of our meeting minutes from 06/06/2013:

	Mihai	**Conrad**	**Guillaume**	**Will**
Implemented	• Design of login, logout and register services with a session timeout • Partial filtering of rogue access	• Multiplayer game logic • Set up stage	Bullet	• Login design
To Implement (by next meeting)	Partial Room Logic	• convert gunship into object • sound design	• Convert bullet into object • Bullet destruction	• Basic websocket support
Ditching	Pure html functionality, in favour of jsp	Paper.js implementation	Unity-based scripts	
To Fix	Further filtering of rogue access	General lagginess	Lag when bullet shot	• Login errors
Fixed				

Implementation Languages

-Java on server-side: it's the required language for writing servlet functionality on the Tomcat server; it has a wide range of packages to support whatever functionality you could fathom (particularly, it has websocket and database access support); should be easy to deploy the webapp on any platform which has the required support (the cross-platform aspect).

-HTML5 with JavaScript on client-side: JavaScript is the de-facto standard for dynamic functionality of web pages on client-side. It is virtually platform independent, and is supported across desktops, laptops, tablets, mobiles etc. In contrast to this, other technologies/languages, such as flash/silverlight/java applets, are not universally supported. This issue occurs with mobile devices, particularly ones running iOS, which is one of the most popular mobile operating systems. It is also the easiest language for a 2D game.

We originally programmed the game in Unity, for its completeness, the knowledge that some of us had in it, and the fact that it is cross platform, but quickly changed because it is a 3D graphic engine that is not at all optimised for 2D games.

Backup system

We have been using a git repository hosted on github.com to maintain our different versions of the project, and to ensure our WebApp is backed up. However, we still had to make sure any sort of merge conflicts would be avoided (since that would have probably resulted in unsatisfactory automated resolution by github.com). The approach was mainly to delegate the tasks concerned with different aspects of our WebApp, with prior agreement on what the requirements would be for each component (as to allow easy integration).

Bibliography and licensing

[1] WebSockets protocol - http://tools.ietf.org/html/rfc6455

kineticJS - http://d3lp1msu2r81bx.cloudfront.net/kjs/js/lib/kinetic-v4.5.3.min.js

Licensed under the MIT or GPL Version 2 licenses.

JQuery - http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.min.js

Licensed under MIT license

createJS - http://code.createjs.com/createjs-2013.05.14.min.js

Distributed under the terms of the MIT license.

MIT License - http://www.opensource.org/licenses/mit-license.html

“The MIT License (MIT)

Copyright (c) <year> <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, **including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software**, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.”

There would be no problems releasing our program, as the MIT license allows us the freedom to distribute all the external software we've used. All assets (sound/graphics) were created by us, so there is no need to worry about copyright in that domain either.

Conclusion

This whole project worked out as expected and we are proud of the results: we implemented a full game logic with AI, with a display, running on a server and linked to a database containing user accounts details and data on it. We also added iPhone controls, which was the main feature we wanted to have originally.

The main problem we encountered was that we did not always communicate effectively the availability of everybody to meet, and sometimes when we met up, we didn't spend time as efficiently as we could because we may not have implemented anything new in the last 3 days, possibly due to working on our other project. If we were to repeat the experience, we would not stick to a rigid schedule of meetings, but rather decide to meet up as and when we needed to; at more appropriate times.

One feature that we wanted to implement originally was a scoring system, and a leaderboard system to compliment it. We eventually found that when we were playing player vs player it is very difficult to score, unless you have a large user base. If player X wins 5 games out of 7 and player Y wins 3 games out of 7 how can we compare them? Player X could've been playing against people vastly inferior to his own standard, whilst player Y could've been playing a world championship game. We decided it would be a better usage of time to implement chat instead, as the game is very social, and captured the enthusiasm of all types of players.

As well as things we would do differently in the future, we would definitely consider our design more carefully from the beginning. Our problem was that we wanted to have a working game, as easy as it could be, to be able to test the first features and then redesign it so that it is easy to extend, if needed. With little knowledge of JavaScript within the team, we started with what we could work out from the Internet, which was mainly anything but nice design. This is definitely something we would not do in the future, and we would rather learn JavaScript to some point and then start coding with the right design of classes, constructors, methods straightaway.