Building a multiplayer space action game in HTML5

Florent Marchand de Kerchove

June 2011

Contents

Acknowledgments

Like any creative project, *Spacewar* was not conceived in a vacuum ; it is not the product of any solitary mind cloistered from outside influences. To the contrary, it was nourished by past experiences, discussions with friends and family, and extensive readings. Even though it is our own, original work, we would like to acknowledge of its most important contributors, be they direct or indirect.

The original idea for Spacewar was inspired by *Slingshot* by Jonathan Musther and Bart Mak. Other inspirations were the eponymous *Spacewar!* conceived by Steve Russel, Martin Graetz and Wayne Witaenem on a DEC PDP-1 at MIT in 1961, and Ambrosia Software's *Escape Velocity* series. Slingshot was introduced to us by fellow students Adrien Bruyere, David Ducatel and Thibaut Demare. Having already toyed with ships and WebSocket, seeing the gravity gimmick of Slingshot initially inspired us to make a browser-based multiplayer version of it.

We would like to sincerely thank fellow hacker Merwan Achibet for continually bringing new ideas to Spacewar and refilling our motivation gauge. Your overnight commits are always a joy to test and fix. Students who helped us play-test Spacewar outside and during classes also deserve recognition, though it might be in their best interest to remain anonymous.

Spacewar was meant to be a side project, not our main semester project at university. Due to the difficulties encountered with our initial project topic, we ended up switching, all for the best. This of course would not have been possible without the understanding of Cyrille Bertelle, our former project adviser, Claude Duvallet, adviser for Spacewar, and Eric Sanlaville, supervisor of our university degree.

These acknowledgments would not be complete without mentioning the authors of the main tools and libraries we used to develop Spacewar. Mentionned in the main text are the JavaScript language by Brendan Eich, CoffeeScript by Jeremy Ashkenas, Node by Ryan Dahl, Socket.IO by Guillermo Rauch. The browsers we use daily to run Spacewar are Firefox by Mozilla and Chrome by Google. The code to Spacewar is managed with the distributed revision control system Git authored by Linus Torvalds, and is remotely hosted at GitHub, which was created by Chris Wanstrath, PJ Hyett and Tom Preston-Werner. Most of these projects are open source, and while we mentioned their initial authors, we must not forget all of their contributors. Lastly, our fingers would like to thank GNU Emacs, for all the hours saved.

1 Rationale

Creating a full-fledged video game is an entertaining way to learn new languages, new libraries, and tackle moderately challenging problems. The more complex the game's rules and features, the harder the problems to solve on the way. In Spacewar for instance, the play field was initially a simple rectangular map. Rather quickly however, running into the map borders was frustrating the players, thus the map was changed to a torus. Having to wrap objects around the map is easy to handle, but doing so seamlessly as to not distract the player requires more ingenuity. Once you add collisions detection around the map borders, then the problem gets really interesting.

Solving such problems in a game setting is satisfying because every time you do so, you can jump into the game and enjoy a new feature or one less bug, leading to a more enjoyable experience. Besides, some problems may lead to elegant solutions, and those are rewarding on their own.

The idea behind Spacewar was to make an engaging multiplayer browser game in a space setting, leveraging current Web standards developments. Being a browser game, it follows a minimalistic design philosophy in order to accommodate the short attention span of the medium. This minimalism is perceptible even in the game pitch, which fits in one line:

> "You are in space. Shoot other ships before they shoot you."

Similarly, the graphics are voluntarily bare, as to let players focus on gameplay, rather than fancy visual effects. This decision also bears the advantage of not requiring strong graphical skills, which we do not possess. Furthermore, the space setting can be used as an excuse for not having any sounds or music. Though it does not imply that every aspect of the game can be justified by the physical laws of space. This is still a video game, and one bereft of any pretension other than unadulterated fun.

The multiplayer aspect of the game was justified by at least two reasons. Firstly, playing a game with human opponents is more engaging than a solitary game, or than a game played against computer AIs. Game AIs have a tendency to be predictable at best, and laughable at worst. While having a fearful AI in the game is a very inspiring problem in itself, we wanted to entertain human players first. Since designing the game mechanics and an AI at the same time could lead to compromises in gameplay to fit the AI capabilities, we decided to favor the mechanics and to take advantage of humans' adaptability. The second reason to have multiplayer was to get acquainted with the WebSocket protocol, part of the HTML5 working standard. This recent development of web technologies allows stable and efficient bidirectional communication between a browser and a server over the HTTP protocol. A

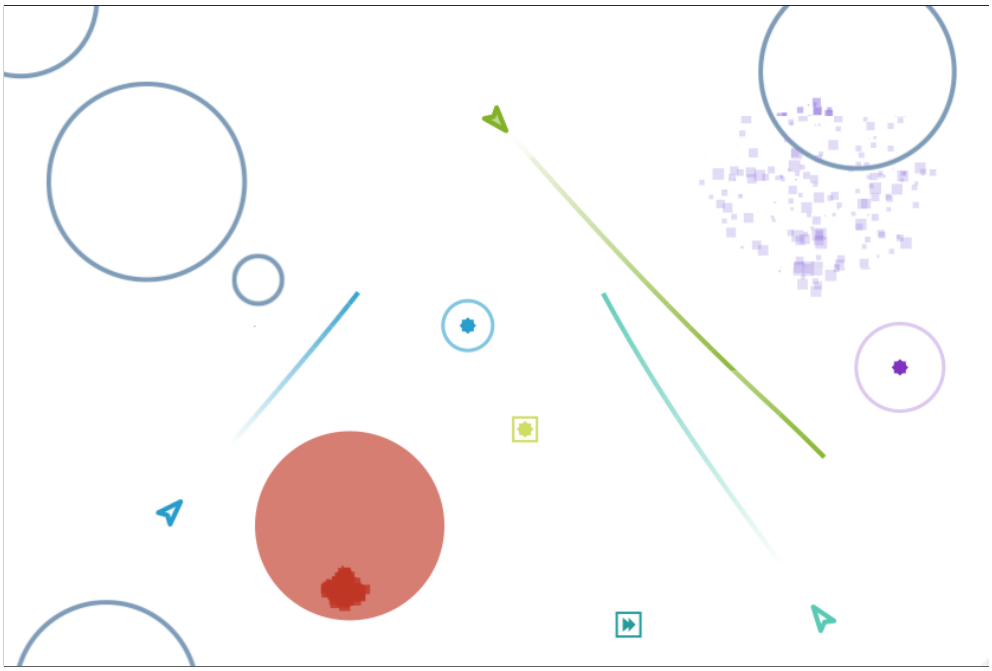

Figure 1: Spacewar action shot. Spaceships battle it out among planets, mines and bonuses.

multiplayer game involves frequent communications between clients and the server: more than fifty updates are broadcasted to clients every second. Therefore, a frantic multiplayer game with WebSockets would stand as a strong testimony of the protocol robustness.

2 Founding technologies

2.1 HTML5 standard

The Web Hypertext Application Technology Working Group (WHATWG) has been spearheading web technology innovation since 2004, branching off the World Wide Web Consortium (W3C) to draft the HTML5 specification when the W3C was solely focused on XHTML 2.0. In 2009, the W3C abandoned XHTML 2.0 and reunited with the WHATWG to formalize HTML5. While the specification has yet to be finished at the time of this writing, some parts are stable and already implemented in most web browsers. The canvas element Spacewar uses for drawing is such a stable part, but the WebSocket protocol is still evolving. It is decidedly rewarding to work with such fresh technologies and to follow the changes they are going through, as the lack of documentation means we have to look up at the specification itself for information, and at browser implementations as well for details.

2.1.1 JavaScript

JavaScript is the programming language of web browsers. Introduced by Netscape for their Navigator, it started off as a modest scripting language used to enhance the possibilities of web designers in the mid 90's. Although it became widely adopted by websites, it was not overly popular with visitors who came to associate JavaScript with cascading pop-ups windows and obnoxious animations. The language gained recognition in recent years, with the advent of Ajax techniques and the maturity of JavaScript frameworks and libraries such as jQuery, CommonJS, or Dojo. Most websites today use JavaScript in a variety of ways: to add visual effects to their design (essentially animations for a more dynamic web), to enhance the site accessibility (visual and oral clues to help visitors with disabilities), or to offer a more solid user interface for web applications (illustrated by most webmail providers).

Fifteen years after its inception, the language has grown to offer a wide array of features, for professional and amateurs programmers alike. Some of those features were inspired by the Scheme language, notably first-class functions, weak typing and closures. The Self language was an other parent, providing the prototype-based inheritance. Although both are idealistic, clean languages, JavaScript syntax is akin to the C and Java family. A convenient choice, since those are still the most popular programming languages, and their syntax is widely recognized by coders of all backgrounds. One last prominent feature of the language is its capability for asynchronous programming without explicit thread handling. Any function can be scheduled for a later execution, without blocking the program flow. All thread management is handled behind the scene by the virtual machine running the program, typically hosted by the web browser. This allows event-driven and concurrent applications to be easily written, since the facilities are built-in.

Still, like every programming language, JavaScript has its downsides. While some are due to historical reasons, notably an initial implementation in ten days by its creator, Brendan Eich, others are deliberate design choices. For instance, declaration of global variables is implicit, which can be the source of insidious errors for unaware programmers.

```
var foo;
function f() {
  foo = 1;
  bar = 2;
  // do something with foo and bar
}
f();  // foo is 1, bar is 2
```

In this code snippet, *foo* is a global declared in the global scope, but *bar* is also a global, although implicitly declared in the scope of *f*. Another quirky language

feature is due to the “==” and ”!=” operators which, like in PHP, do implicit type coercion. On top of being inefficient (strings have to be converted for numbers and vice versa), this coercion voids the transitivity of equality. The following example illustrates this problem:

```
"" == 0     // true
0 == "0"    // true
"" == "0"   // false
```

Fortunately, strict equality (and inequality) operators are also provided and behave rationally.

In light of these peculiarities, seasoned JavaScript programmers established a guideline of recommended JavaScript style which emphasize clear and safe JavaScript programming practices [14]. Popular frameworks also tend to enforce these practices and have their users follow them in order to avoid errors that are hard to track. Another solution to alleviate these downsides is to use another syntax that compiles directly into JavaScript, bypassing the error-prone features.

CoffeeScript is a language by Jeremy Ashkenas [1] that does just that. It compiles directly into JavaScript without any overhead at run-time. CoffeeScript borrows its syntax from Ruby, another popular programming language in the web development community. This syntax is characteristically clean: parentheses are optional, and indentation delimit blocks instead of accolades. In addition, CoffeeScript provides syntactic sugar for idiomatic operations like iterating over an object properties, array comprehensions, and lambda function declaration. Consequently, CoffeeScript code is more expressive than the operationally equivalent JavaScript code, and time is saved both when programming and reading CoffeeScript code.

Even though Spacewar began by using JavaScript, the decision to switch to CoffeeScript was easy to make for all the preceding reasons.

2.1.2 Canvas element

The canvas HTML element [6] is an important part of HTML5. It was initially created by Apple for Safari and the Mac OS X Dashboard, but is now implemented in all the major web browsers [1].

The intent is to provide an area on web pages upon which to draw freely, as an alternative to vector graphics provided by SVG (Scalable Vector Graphics). While SVG already allowed scripted animations to be run freely on web pages since its initial release in 2001, SVG images have to be inserted into the DOM tree, and this can be a significant computation overhead if many SVG images are inserted and removed from the DOM tree. When fast animations are required, this overhead is unacceptable. Being a single DOM element, the canvas provides a potentially faster way to

[1] By major web browsers we mean Firefox, Chrome, Internet Explorer, Safari and Opera.

draw animation than SVG. But these two graphics solutions are not interchangeable.

First, the canvas element is bitmap-based. This means that graphics drawn on the canvas are resolution dependent, whereas SVG animations are freely scalable. Bitmap graphics are faster to process, thus well suited to animations that require a high number of frames every second, like games. Scalable graphics are great for everything else, since their image quality do not degrade when scaled at any size.

The canvas element also has the ability to use a 3d rendering context called WebGL [10], enabling OpenGL applications to be embedded in web pages and rendered using a software OpenGL implementation, or even a hardware one if the proper driver is present. Implementations of this 3d context in major browsers is ongoing. The latest Firefox, Chrome, Safari and Opera browsers support WebGL, but not on all platforms. Besides, since calling the GPU driver directly can lead to crashes unrelated with the browser itself (caused by faulty drivers or poor graphics management in the OS), WebGL support can be deactivated by default in some browsers, or for unsupported hardware. Nonetheless, some demonstrative applications already exist, with the most impressive being ports of famous 3d games like a Quake 3 level loader [8] or an interactive film by Chris Milk [11].

Finally, should web applications using canvas require even more processing power for their animations, major browsers are beginning to provide hardware acceleration even for the 2d drawing context. This is also optional and enabled client-side, due to the stability issues mentioned above, but still is a nice option to have when high performance is a concern.

The consequence of providing all those facilities to web scripts, both versatile with SVG and powerful with the canvas element, is the diminishing need for proprietary alternatives like Adobe Flash or Microsoft Silverlight. A few years back, Adobe Flash was the *de facto* standard for web animations. Today, with browsers adopting HTML5 standard and especially the canvas element, animations can both be fast and portable. Furthermore, the specification is fully open, as are implementations in most browsers.

Spacewar makes full use of the canvas element. All the game is currently drawn on it, and it fills the entire web page. Some SVG images are also used for the more static configuration menu.

2.1.3 WebSocket

The major part of Spacewar is its multiplayer aspect: having multiple players participating in the same game, each using a different browser on their own computer. This is enabled by the introduction of WebSockets to HTML5.

The HTTP protocol was conceived as a unidirectional protocol: from the client to the server. The server can not initiate an unsolicited connection with a client,

and is only allowed to send data to the client in response to a previous request from the client. Yet, asynchronous updates from the server have become a major part of today's web browsing. The usual solution is to abuse the protocol in part, by having the client frequently poll the server for updates. Typically, the client would send a request to the server every two seconds, and the server would respond with update data, or with an empty response if no update occurred. This is a simple way to provide the illusion of asynchronous updates, but it comes at a price.

To receive update in a timely fashion, the polling frequency should be short, around five seconds. This means that every five seconds, a HTTP request is sent to the server, and a response is received. Sending a new HTTP request requires establishing a TCP connection with the server, and that in turn takes some round trips between the client and the server. In addition, the request and response have to contain HTTP headers that add to the packets' size, though the information they transmit is mostly the same each time. This short polling technique is thus very inefficient, as not only time is wasted by setting up a full TCP communication each time, but bandwidth is also squandered.

To alleviate these problems, more refined techniques have come to light. The two most common mechanisms are known as HTTP long polling and HTTP streaming. Both are described in RFC 6202 [9]. As the name implies, long polling consists of sending a request to the server, who will delay its response until there is data to transmit. When the client receives the response, it immediately sends a new request for the next update. In HTTP streaming, the server sends its response in parts, thus keeping the connection with the client alive, until its renewal after a set amount of time. While more efficient than short polling, both have disadvantages over a straightforward TCP socket. When renewing the long poll request or stream, messages can not be sent from the server, and must thus be buffered until the connection is established. Other difficulties may arise when intermediaries (proxies, gateways) are present between the client and the server, as they may decide to cache the server responses, thus defeating the mechanism. Best practices for implementing these techniques are described in the RFC.

A better solution is to augment the HTTP protocol, allowing true, persistent bidirectional communication without much overhead. That is precisely the high-level description of the WebSocket protocol [4]. This protocol consists of establishing a TCP connection between a client and a server, allowing two-way message passing after a HTTP handshake part. Using the HTTP request/response model, both client and server acknowledge of a "HTTP upgrade" to use the WebSocket protocol. Once they agree, messages can be freely sent between both entities until the connection is closed. In addition to the TCP connection, the WebSocket protocol provides other features. To quote the protocol draft:

- a Web "origin"-based security model for browsers;

- an addressing and protocol naming mechanism to support multiple services on one port and multiple host names on one IP address;
- a framing mechanism on top of TCP to get back to the IP packet mechanism that TCP is built on, but without length limits.

The WebSocket protocol enforces the same origin policy commonly used in web browsers. Basically, WebSockets established between a browser and a web server at http://www.foo.com/ can only be accessed by scripts running for a page from a server with the same domain. Scripts from other domains do not have legitimate access to resources (scripts, sockets) of others.

The WebSocket API [7] offered to web browsers scripts is rather straightforward, and event-based. In simple setups, the client only has to create a WebSocket object, then registers function callbacks for the following events: connection open, message received, connection closed. This WebSocket object can be used to send messages to the server, and to close the connection. Messages can either be sent as UTF-8 strings or as raw binary.

2.2 Node

Obviously, WebSocket support is required both on the client and on the server. On the client side, a check may be done when the game script executes to detect WebSocket support in the browser, and act accordingly. On the server front, the choice is more limited. The early prototype of Spacewar used a PHP implementation of the WebSocket protocol on the server. Back then, the server was only used to pass messages around between clients. All the game logic was handled by the clients themselves, and each client sent its state to all the others, via the server. It quickly became clear however that this PHP implementation of WebSocket was meant for trivial demonstrations purposes, but not for real applications.

As a result of the WebSocket protocol being recent and not finalized, the number of implementations available outside of web browsers is not overwhelming, though sufficient. Although the protocol is not overly complicated, implementing it was beyond the scope of Spacewar. Besides, due to security concerns, the protocol is still evolving. An implementation that can stay up to date with current and future versions of the protocol is preferred. This essentially means that the chosen WebSocket implementation should be popular enough to ensure that it will be supported until at least the protocol finalization. A rapid search indicates there are already quite a few implementations for C, C#, Java, Ruby, JavaScript, and more. The JavaScript implementation is of particular interest, and was chosen for reasons we will now expose.

First and foremost, since JavaScript was required for programming the client, having it as the server language helps promoting code reuse and consistency. The code is clearer as a result, since some client constructs are mirrored on the server.

Secondly, the chosen JavaScript implementation of the WebSocket protocol is actually a module for a high-performance server back-end software running JavaScript code, called Node (or node.js) [3]. At the core, Node is an abstraction of asynchronous I/O with a layer of web-oriented networking facilities designed for building scalable web servers. Under the hood Node is running Google's V8 JavaScript engine, which allows users to program all their server code using only JavaScript, although Node itself is essentially made in C++.

With the growing number of Internet users, it is not uncommon for web hosts serving popular content to handle a million or more daily hits, and a hundred thousand of simultaneous clients. Against such numbers, very efficient server software (and hardware) is required. There are three main approaches for serving content at this rate in server software: caching, multithreading and asynchronous (non-blocking) I/O. Caching is very useful for static content, but not for a highly dynamic application like Spacewar. While allocating a thread for each client (or a group of clients) and retrieving content with blocking I/O in each thread is feasible, implementations of threads in most systems makes this solution sub-optimal. For one, managing ten thousand threads can become quite complex. In addition, the costs of thread allocating, context switching and scheduling hamper this method's scalability.

On the other hand, asynchronous I/O is rather straightforward: instead of spawning a new lightweight process (thread) to read and send a file to the client, just wait for the system to signal the file readiness, and execute a callback function to read and send it. The cost of handling one more client is much lower than for multithreading, although not all non-blocking I/O mechanisms in the OS are equal. Hardware interrupts are favored, but might not be available, in which case the slower method of polling is used. Nonetheless, non-blocking I/O has garnered a strong following in the web development community, assessed by the popularity of server software like Python's Twisted, or Ruby's EventMachine. This is explained partly by the popularity of the language they are implemented in, partly by the performance they provide, and mostly by the convenience of their event-driven model for programmers.

Since all the work with asynchronous I/O is done in callbacks, this directly translates into programming for events, a recurrent pattern of web programming. For instance, an HTTP server will have a function to handle a request event. Every request sent to the server will wake up Node, trigger this function, and send Node to sleep once the function has returned. It must be noted that since no threading is involved, the callbacks execute sequentially rather than concurrently. In particular,

callbacks that are slow to return will become a bottleneck for the server scalability. Nevertheless, the event-driven approach is quite fit to the HTTP request/response model, as well as other network applications.

Node also aims to be very modular: even core features are provided as modules lazily loadable in a server program. The WebSocket protocol is available in Node thanks to such modules. Spacewar uses the one named Socket.IO [13]. While other modules expose the bare WebSocket protocol in Node, Socket.IO can fallback to other protocols (like Ajax, HTTP long polling or HTTP streaming) if the client has no support for WebSocket. This is highly convenient, given the discrepancies between browsers as well as between a different versions of the same browser.

3 Other works

Spacewar arose from our want to design and play a space action game leveraging modern web technologies. Outside of the gravity gimmick inspired by the open source game Slingshot [12], there was no direct influence by other works until we took a peek at what was available.

Eerily enough, the space setting seems to be popular, as at least two games built with JavaScript and Node bear the same premises. The first one is Lazeroids [2]. That game was initially conceived using Ruby on Rails over a week-end competition, and later ported over to Node. The gameplay is a voluntary homage to the classic 1979 arcade game Asteroids where a lone spaceship fires round-shaped bullets at dangerously concave asteroids drifting in the otherwise empty space. Lazeroids allows multiple players to play in the same world and features a score board. Attacking other players did not seem to work right when we tried it, and the gameplay also feels dated, with reason. Still, one must keep in mind that Lazeroids was conceived as a demonstration of Ruby on Rails and HTML5 capabilities over a 48-hours period.

The second space-inspired browser game is Rawkets [5]. Here the focus is on player versus player combat, as there are no other interactions with the world. The fighting stage is delimited by a thick rectangular wall, and the combat is very basic. Ships fire bullets that go straight ahead dealing damage to the first target hit. Ships can sustain some damage before exploding, and recover damage over time. The graphics are elementary, except for the somewhat fancy static background and interface. There are even sound effects and a background music [2]. Like Lazeroids, Rawkets primary purpose is to demonstrate the capabilities of HTML5 and Node in the fun setting of a multiplayer space fighting game.

[2]Rawkets uses Adobe Flash for sound. Although the HTML5 standard introduces an audio API, browser implementations might have been lacking when Rawkets was conceived.

4 Inner workings

4.1 Overview

4.1.1 The game's rules and goal

As previously stated, the intent in building Spacewar was to become more comfortable with standard and future web technologies, by creating a simple, yet engaging multiplayer game. The core gameplay revolves around shooting spaceships in space, with the added twist of compensating for the surrounding planets' gravity that draws spaceship fire.

Browsing to the game server URL immediately jumps the client into the game. Each player controls exactly one ship. The commands are simple: ships can rotate left or right, thrust forward, fire a bullet or use a bonus. Ships are brittle: one hit is all it takes to destroy them. Obstacles are plenty: planets, bullets, other ships, and lethal bonuses. The bright side is that dying bears no in-game penalty aside from losing any held bonuses, as hitting the spacebar immediately spawns a new ship. Hearing the opponent gloat over his victory is punishing enough.

The action takes place on a rectangular map wrapped at the edges, effectively simulating a torus. A plain rectangular map with border walls is not symmetric: corners are notably disadvantageous for players who wander in them, since it is much harder to escape from them. The central area will thus see most of the action. A torus is symmetric action-wise, as all area have the same mobility. The toric map is harder to handle though, especially when considering collisions at the map borders, or when drawing it seamlessly in the client.

The map is populated with immobile planets, which have two roles in the game. First they act as obstacles for players, as a ship colliding with a planet will immediately turn to sidereal dust. Players can not blindly go forward, but must learn to maneuver skillfully between cluster of planets in order to survive. Secondly, planets affect the trajectory of bullets fired from the spaceships. Each planet has a gravity parameter proportional to its radius, and bullets are subject to every planet gravity field. Larger planets pull harder on bullets, eventually crashing them on their surface. Players have to learn how bullets react to gravity, as it is crucial to improve their aim but also to better dodge other ships' bullets.

The game has no explicit goal other than enjoyment. Adding some kind of scoring system, based on the number of ships destroyed and length of survival would be trivial, but whether this is effectively beneficial to the game experience is still undetermined. The core formula has proved to be satisfying enough for the time being.

The bonuses add variety to the game, by introducing new obstacles and weapons. New bonuses are regularly dropped onto the map, staying at their location until a ship picks it up by flying over it. Ships can only have one bonus at a time. Flying

over a bonus when already holding one will replace it. Once picked up, a bonus can be used until it has no more charges. Most bonuses currently implemented have only one charge. Bonuses are not definitive, and need testing to determine if they have their place in the game. The upside is that bonuses are easily added and removed from a game for test purposes using parameters. Player feedback can be used to increase or decrease the probability of a certain type of bonus appearing. Ideally, these parameters could be altered even during a game, matching the interests of current players. The most successful bonus so far is the mine, immobile when placed on the map, but which explodes when a ship or a bullet enters its detection radius. These mines are a useful to dispose of tailing opponents, or to set up traps in tight planet clusters.

4.1.2 The decentralized prototype

Spacewar is divided into a client and a server program. Both are written in CoffeeScript, compiled to JavaScript before being run. The client program is intended to be executed inside a web browser supporting the HTML canvas element. The server must be run by Node with additional modules installed.

Historically, the initial prototype of Spacewar was meant to work without a server. The client handled all the game logic simulation, in addition to drawing, and one could play the game even if a server was not available. On the other hand, if a server was running, then clients would send it their position and bullets, and the server would broadcast them to all other connected clients. The server thus acted as a mere relay.

This decentralized model had its merits: clients could play without a server, nearly all computation was offloaded to clients, thus greatly alleviating the server load, and the server was truly scalable, able to handle many clients since its only role was to coordinate packets between clients. But this approach was not devoid of problems. The first one was the game state synchronization across all clients. Since all clients ran their own game logic and only sent updates concerning their own actions, the game state was different for each one ; there was no authoritative state as in a centralized model. This allowed asymmetric situations to arise, where a ship would be dead for some clients but not for others. Although this might have been solvable, by acknowledging collisions between entities for affected clients for instance, the far greater issue of cheating remained.

One thing to keep in mind with secure web development that also applies to games, is to never trust the client. The client, when given the chance, will always try to exploit any flaw to gain advantage. In a game setting, this amounts to cheating, which ruins the game experience for everyone else involved. This is not to say that all clients should be regarded as evil exploiters, as most are not, but care should be taken as to minimize, and even annihilate the risks. If by any means someone is able

to gain an unfair advantage, then, given enough time, someone will. Manipulating the game code is made even easier with JavaScript browser games: no decompilation is necessary as all the code is interpreted and embedded in the web page. With this in mind, letting clients make unilateral game decisions is ripe for exploits. In the Spacewar prototype, clients could easily change the ship maximum speed, to move faster than other players and gain advantage. The sole answer is to check all data received from clients, and drop faulty packets.

Again, cheating is avoidable, game state synchronization is doable, but complexities and subtleties quickly arise when trying to solve both problems. The centralized model of a game server handling all game logic and broadcasting it to clients is far simpler to design. The choice was thus made early to switch Spacewar to this centralized model, at the cost of losing the ability for clients to play without a server, and greatly diminishing the scalability, but working as expected.

4.2 Client side

The client program has two roles:

- It gathers keyboard input from the player and forwards them to the server.
- It receives updates from the server and renders the game to the canvas.

In a way, the client can be thought of as a terminal with a fancy interface.

First, it has to establish the connection to the server. This amounts to creating the WebSocket, and waiting for the connected event. When this event is received, it contains an id number used to identify the client, which the client saves. Then, the client's preferences (name and color of ship) are sent, if any, and a ship is requested. Once notified of the ship creation, the game render loop is started, and the client can play.

From then on, every time the client hits or releases a key, a message is sent to the server. Knowing which keys each client has pressed, the server updates each ship accordingly when going through its own update loop. When any entity of the game world changes, the client receives a message containing the new values to synchronize its local state with the server state. The client is only told what it needs to know in order to draw the game world correctly. Information unneeded for drawing is never transmitted. Simultaneously, the rendering loop displays the game world to the player, centered around its ship.

4.2.1 The rendering algorithm

Drawing the game is the main role of the client, and the most expensive in computer time. All drawing takes place on the HTML canvas element, using the 2d rendering context. The canvas is stretched to fill the client's whole window, adapting

to eventual resizing events, thus maximizing the player's view. To ensure smooth rendering, the scene is requested to be drawn at 60 frames per second. This is only a request: clients will do their best to reach this frequency, but under-performing clients might not reach it. In this case, the rendering will be choppy, and playing might become difficult, or even unsatisfactory. Efforts should thus be made to ensure most computers can draw the game fast enough to allow a smooth play. Optimization comes after correctness however, and since Spacewar is not feature complete at this stage, busy action can lead to slowdowns on even recent hardware.

The rendering loop algorithm is very similar to the following code:

```
redraw = (context) ->
  context.clearCanvas()

  # Center view around the player's ship.
  centerView()

  # Draw all objects.
  for obj in gameObjects
    obj.draw(context) if obj.inView()

  # Draw all visual effects.
  for e in effects
    e.draw(context) if e.inView()

  # Draw outside of the map bounds.
  drawInfinity(context)

  # Draw user interface.
  drawUI(context)
```

We start by clearing the whole canvas, which contained the previous frame. This is needed since the scene is centered around the player, thus every object in view has to be redrawn every time the player moves. On this blank canvas, the game is drawn in layers. Each layer is drawn atop of the previous one, and any drawing done in a layer obscures the drawings done at the same place in lower layers. The first layer contains every game object (ships, bullets, planets, bonuses). The second layer is filled with cosmetic effects that are tied to a particular object and exist only on the client. Ship explosions are the only effect present so far. The final layer is used for the user interface: radar symbols drawn at the window's edges to indicate other players and incoming bonuses that are out of view. The drawInfinity method warrants a deeper explanation.

4.2.2 Drawing infinity ... and beyond

The purpose of the drawInfinity method is to render the toric map to the player by redrawing each visible object outside of the map edges. Without drawInfinity, a player near an edge of the map would not see the objects beyond the edge until he crosses it and is wrapped around on the torus. This method helps to create a seamless transition when wrapping around the edges. The following diagram illustrates this.

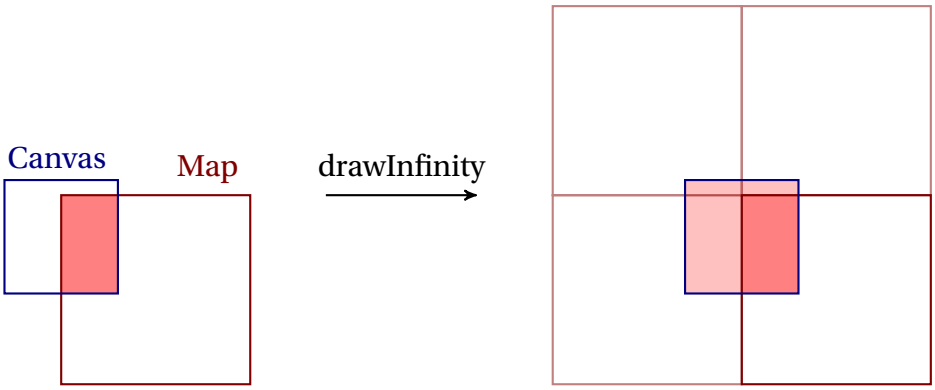

Figure 2: The effect on drawing the map on the canvas with the drawInfinity method. Without drawInfinity, the canvas has blank areas when the player is near the map edges. With drawInfinity, the whole canvas is filled with copies of the map, seamlessly rendering the torus.

On figure 2, the game screen is shown without calling drawInfinity (left-hand part), then with calling drawInfinity (right-hand part). Without drawInfinity, the visible part of the game map is drawn at the right of the canvas, leaving the area outside of the map blank. Since the map is a torus, there can not be any undrawn part on the canvas. Here the left part of the canvas should show the rightmost area of the game map, the top part should show the bottom of the map, and so on. That is precisely what the drawInfinity method does: it redraws the map at the edges of the original map by translating it. The method collects the edges currently visible by the player and draws the map for each one. The results are witnessed on the right of figure 2, where the map is cloned to fill the whole canvas.

While this drawing method ensures the world is correctly rendered as a torus, there are additional details to take care off. First, all objects should behave accordingly to the toric condition of the map. That is the server's role when updating the game world. For instance, bullets should wrap around and be affected by the gravity of all surrounding planets, even planets that are beyond the map edges. Another example is given by the planets: they should not overflow the map when created, otherwise they would be rendered as overlapping another planet from the other side of the map, which is not allowed. Last but not least, the radar that is used to

show other players' ships and incoming bonuses that are out of view must select the nearest target among all the "ghosts" of an object, those mirror images drawn by the drawInfinity method. When the player's ship is near the left border of the map, and another ship is near the top right corner, the radar should indicate the shortest route to the target, which is realized by going left and up, wrapping around the edges, rather than traversing the whole map by going to the right (figure 3).

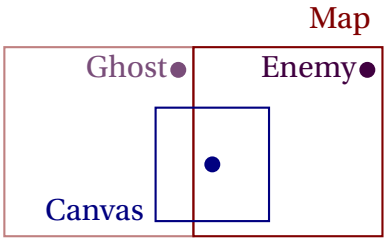

Figure 3: Finding the nearest "ghost" in all clones of the map. Here the player is the blue dot at the canvas center, and the enemy at is at the top right corner of the map. Since the map is a torus, the quickest route to the enemy is by going left and up, toward the enemy "ghost". The player radar always indicates the nearest ghost rather than real map position.

4.2.3 Rendering performance

Drawing to the canvas at 60 frame per second, even in 2d, can tax even recent hardware. Speeding up this step is beneficial to gameplay, as smooth rendering is essential to fast paced action-oriented games. While we prefer to avoid premature optimization, some has already occurred and bore great benefits.

The first optimization is avoiding to draw objects that are out of view. Each object is drawn only if it can be seen by the player, within the bounds of its screen (there is no line of sight restriction). This saves CPU cycles, as even if a pixel would not be rendered to the screen, browsers still take time doing operations on it. That is because the canvas element can be drawn upon even if it is not attached to the DOM tree, saved for later purposes. Since the map can be vastly larger than the player's view, and full of objects rather costly to draw, this check is judicious.

The second optimization is avoiding to redo costly operations that can be saved. Planets are immobile objects that never change their shape or color during the game. They are also numerous on the map, thus rather costly to draw, even though they are represented by bare discs. By drawing each of them only once to a devoted canvas, we can later draw this hidden canvas onto the real canvas presented to the player at a lower computational cost. Drawing to a hidden surface beforehand and applying this surface to the game canvas like a stamp is a cornerstone technique of 2d game programming called *spriting*.

These optimizations already proved their worth by stabilizing the framerate, but it must be noted that the low-level tweaking common in video game programming is restricted in this setting. The programmer has only access to the canvas, and the browser is in charge of the lower-level operations and interaction with the graphic capabilities of the client's machine. While tuning for specific browsers is feasible, tailoring the game to the features offered by a particular CPU or GPU is beyond reach. This is the cost of using a scripting language running inside a browser. Luckily, implementers of the canvas element are committed to minimize this cost. For instance, Gecko (used by Firefox) WebKit (used by Chrome and Safari) engines and the latest Internet Explorer provide optional hardware acceleration of the 2d rendering context. In addition, they include an implicit double buffering of the canvas: drawing operations are done off-screen and the canvas is refreshed only once when the script is done drawing the frame. This is quicker than refreshing at every drawing operation.

4.3 Server side

The server program is where the game actually resides, where game logic, collision detection, collision resolution, and synchronization between clients arises. As previously stated, the server is written in JavaScript (compiled from CoffeeScript) and run in a Node environment.

Before launching the actual game server, some preparations are in order. First, a HTTP server is started to serve the client files. The second step is to bind the WebSocket to the HTTP server, listening for a HTTP upgrade request initiated by the client program. Finally we setup the callbacks for the client connection, disconnection and message events, initialize the game map and then launch the game loop.

4.3.1 Client-server communication

When a client connects to the server, a player id number is established, an associated player object is created server-side, then the client is notified of the connection. Following that, the client should request a ship, in which case the server will create the ship and send a full game update to the client containing all game objects. Once this is done, the client can play.

During the main course of the game, the only messages received by the server from clients are input related. Pressed and released keys are sent to the server, which updates the corresponding player object.

In the event of a client disconnection, be it voluntary or accidental, other clients are notified and resources are freed.

4.3.2 Setting up the game map

To initialize the map, its dimensions are first retrieved from the preferences file. This file contains constant values used throughout the game simulation which can be customized to tune the game mechanics. For example, the preferences file describes the maximum allowed ship speed, the intensity of the gravity effect on bullets, and the activation time of mines.

Once the map size is known, we must populate it with planets. The number of planets to place and their radius range is also loaded from the preferences file. Then each planet is randomly put on the map, provided that it does not overlap any previously put planet. There is a chance for each planet to have an accompanying satellite. When this happens, the total radius of the system is taken into account when checking against overlaps. Satellite size, rotational speed and distance to host planet are all parameters in the aforementioned file.

4.3.3 The update loop

The most run code on the server is the one called by the update loop. Similarly to the client drawing loop, the server update loop is run at a high frequency: every 20 milliseconds, which translates to 50 updates per second. The client and server update loops do not have to be synchronized, since there will always be an added network latency between them. The server should update very often though, to be able to quickly respond to user input.

Let us have a look at the (abridged) update loop:

```
update: () ->
  # Process input from players.
  for id, player of players
    player.update()

  # Move all objects and update their grid position.
  for id, obj of gameObjects
    obj.move()
    if obj.tangible()
      placeObjectInGrid(obj)

  # Check and handle all collisions between objects.
  handleCollisions()

  # Let objects update and record their changes.
  allChanges = {}
  for id, obj of gameObjects
```

```
    obj.update()
    allChanges[id] = obj.changes()

  # Send only the changes to all clients.
  socket.broadcast
    type: 'objects update'
    objects: allChanges
```

The first step is to process the input of each player. Spacewar requires only five keys:

- Up arrow to thrust forward,
- Left and right arrow to rotate,
- Spacebar (or A) to fire,
- Z to use the carried bonus.

The client sends its keys to the server which saves them for this update purpose. When processing each player's input, the server only has to check whether a key is pressed to update the player's ship accordingly. For example, if the left arrow key is pressed at the time the server enters the update loop, the ship's facing angle will be decreased [3].

The next step is to update all objects. This is divided into three parts: first objects are moved, then all collisions between objects are checked and handled, and finally objects have their state updated. With this division, objects can post-process collisions in their update method instead of requiring a separate method with duplicated code. Moving all objects is simple: the position is updated with respect to the object velocity. The only subtlety is to wrap around the map edges. Some objects like planets and bonuses do not ever move. Bullets are of interest since they are affected by gravity from planets: a Newtonian gravity formula is applied to the bullet acceleration vector for every planet around.

Once positions are updated, objects are placed in a grid used to check collisions. We will cover collisions in further details in 4.3.4. After collisions are processed, the state of each game object is updated. In this step, objects can update anything not related to position, which is handled in the position update. For example, mines grow their detection radius and satellites increase their rotation angle.

Most objects will have their state changed as a result of these steps. Moving changes the position vector, mines change their hit radius at each update, ships

[3] The origin of the HTML canvas element is at the upper left corner, with x increasing to the right and y increasing to the bottom. Consequently, angles of the unit circle increase clockwise instead of conventionally increasing counterclockwise.

and bullets can die after hitting another object, etc. Clients should be notified of all these changes, but there is no need to transmit fields that have not been modified. Full game objects are already sent to the client at connection time. Further updates only transmit the changes to avoid wasting bandwidth and unnecessary serialization. For this purpose, all game objects have the possibility to mark fields to be watched for changes. Changes to these marked fields are recorded into a dedicated object that is gathered in the update loop by calling *obj.changes()*. When changes from all objects are obtained this way, clients are notified of the game update by broadcasting.

4.3.4 Handling collisions

Treating collisions between game objects is straightforward, except for a few subtle details. The basic principle is to check every couple of objects for collisions and handle side effects with respect to object type. In Spacewar, all ships would check if a collision occurred with any other ship, bullet, planet, bonus, etc. Obviously a collision is symmetrical, meaning we only have to check half of the couples. Once a collision is detected side effects are applied. If a collision occurred between a ship and a bullet for example, the ship would explode and the bullet would enter its dead state. Both would not be able to collide with another object anymore. On the other hand, if a ship collided with a planet, the ship would still explode but the planet would be unaffected. Processing side effects from a collision at the same time for both colliding objects is clearer, since all effects from the collision are in one place. It is also easier than having objects handle the collision themselves, where issues related to the absence of atomicity arise.

Accurately checking for collisions between two objects often requires solving equations, which can be quite costly depending on the shapes involved. Accuracy can be traded for speed by using approximated equations. An uncompromising speedup can be obtained by only checking collisions between nearby objects. To group objects by proximity, any form of spatial hashing can be used. Spacewar elected the spatial grid approach, as it is straightforward and fit to the 2d rectangular map. The map is divided into same-sized regions at its creation (see figure 4). During the update loop, when objects are moved, they are inserted in all regions they overlap with. Collisions are then only checked between objects belonging to the same region. Due to the toric nature of the map, the spatial grid has to be toric too. Objects near the map edges can lie in grid regions adjacent only by wrapping around. Collision checks thus have to work on parts rather than whole objects. Other optimizations rely on ensuring game objects are tangible before diving into costly computations.

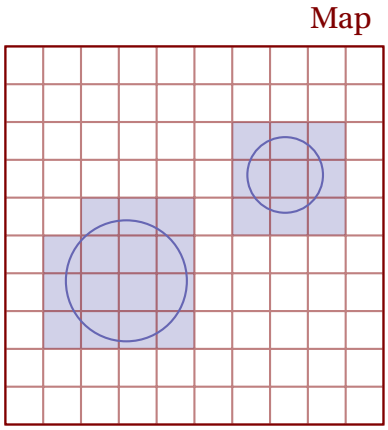

Figure 4: The map is divided into same-sized cells and collisions are checked only for objects belonging to the same cell. Objects are inserted into all cells they overlap with. Here the two blue planets are inserted in all cells filled with blue.

5 Future improvements

In the current state, Spacewar is playable, enjoyable and rather stable. However, there is still room for improvement. On the short term, we would like to allow clients to launch games and invite friends in it. The envisioned scheme is the following: after landing on the Spacewar homepage, the client would be greeted with two choices of playing straight away, and creating a new game. Playing straight away would jump the client into a randomly determined active game. The client could also express wishes on its playing preferences, like specifying the maximum number of player in the game, the density of planets, the map dimensions and so on. Those wishes would narrow the search for an active game to join. Alternatively, were the client to choose to create a new game, he would be prompted for more in-depth settings, similar to those currently present in the preferences file. He would be able to set the maximum number of players allowed in the game, the map dimensions, the density of planets and satellites, but also the allowed bonuses and bonuses timing. Eventually, he should be able to tweak the game settings so much that the created game would have a unique quality to it. Once he is satisfied with these settings, the client can start the game. An active game would have a unique URL attached to it. The client who created the game should forward this URL to all the players he wants to spar with. Upon accessing this URL, those players would immediately join the action.

On the technical side, this scheme of having multiple games running concurrently would be interesting to implement. This might need some more thought, but at this time we envision to have a delegated HTTP server program running in front of the multiple Spacewar games, forwarding the messages from clients to dedicated

game processes, depending on the access URL. This program would thus act similarly to a reverse proxy.

In order to allow at least a dozen of those games concurrently running on the same machine, each game hosting from a handful to potentially ten or twenty players, the server program needs to only consume a fraction of the machine's resources. Otherwise the hardware costs to support even a hundred concurrent players will quickly rise to the unaffordable level. Consequently, optimization in the server is another short term goal. This optimization should focus on scaling resource usage with the number of game objects. It goes without saying that optimizing client drawing is another priority. The game is currently rather demanding, even though the graphics are bare. This is due in part to the immaturity of implementations of the canvas rendering context in current web browsers. Nonetheless, there are certainly ways to fasten the rendering loop without waiting for optimizations on browsers to happen.

Other priorities, on a longer term, are the compatibility and stability of both client and server programs. For example, every browser handle input a different way: we need to account for this fact. The goal is to provide a similar experience on all supported browsers. Particularities in each browser makes this a challenging task. For instance, Firefox does anti-aliasing on the canvas element, while Chrome does not. This translates into the impossibility to specify sub-pixel coordinates to draw at when using Chrome. Since anti-aliasing is not part of the canvas element specification and delegated to implementers, we have to accommodate these peculiarities.

More gameplay-oriented features are planned as well: more bonuses, single-player action, and some way of tracking progress in multiplayer matches. We might also consider making the client compatible with touch devices, both to widen the audience and to learn how to interact with those devices in a web setting.

Be it a new language, an experimental library, a novel algorithm, learning has always been the strongest motivation to take on this project. We picked up quite a few skills along the road, and will continue to do so until we run out of ideas to improve Spacewar. Then we will acknowledge of the game maturity, rest a while, and move on to another project, ready to learn anew.

References

[1] Jeremy Ashkenas. *CoffeeScript*. 2009. URL: http://coffeescript.org/.

[2] Huned Botee et al. *Lazeroids*. 2009. URL: http://www.lazeroids.com/.

[3] Ryan Dahl. *Node.js*. 2009. URL: http://nodejs.org/.

[4] Ian Fette. *The Websocket Protocol Standards Draft*. 2011. URL: http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-07.

[5] Rob Hawkes. *Rawkets*. 2010. URL: http://www.rawkets.com/.

[6] Ian Hickson. *HTML Living Standard: the Canvas Element*. 2011. URL: http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html.

[7] Ian Hickson. *The WebSocket API*. 2011. URL: http://dev.w3.org/html5/websockets/.

[8] Brandon Jones. *Quake 3 WebGL Demo*. 2010. URL: http://media.tojicode.com/q3bsp/.

[9] Salvatore Loreto et al. *Known Issues and Best Practices for Long Polling*. 2011. URL: http://tools.ietf.org/html/rfc6202.

[10] Chris Marrin. *WebGL Specification*. 2011. URL: http://www.khronos.org/registry/webgl/specs/latest/.

[11] Chris Milk. *ROME, "3 dreams of black"*. 2011. URL: http://www.ro.me/.

[12] Jonathan Musther and Bart Mak. *Slingshot*. 2007. URL: http://slingshot.wikispot.org/.

[13] Guillermo Rauch. *Socket.IO*. 2010. URL: http://socket.io/.

[14] Ivo Wetzel and Zhang Yi Jiang. *JavaScript Garden*. 2011. URL: http://bonsaiden.github.com/JavaScript-Garden/.