Universitat de Girona
Escola Politècnica Superior

Projecte – Treball final de carrera

Estudi: ETIG

Títol: Node.js i html5 per al desenvolupament de jocs

Document: Memòria

Alumne: Daniel Jordi Perez Quilez

Tutor: Ignacio Martín Campos
Departament: Informàtica i Matemàtica aplicada
Àrea: LSI

Convocatòria (mes/any) Juny/2015

Índex

Universitat de Girona

1. Introducció, motivacions, propòsits i objectius del projecte

En aquests darrers anys, el món web ha evolucionat considerablement amb el canvi de HTML 4 a HTML 5. Hi ha moltes característiques implementades, per aquest nou estàndard que han ampliat molt el ventall de possibilitats que ofereixen actualment els navegadors moderns. Algunes d'aquestes són: websockets [13], canvas [15], webgl [16], webworkers [14], creació de nous tags de html i modificacions d'alguns ja existents, entre d'altres. Les que he fet servir en el projecte són canvas, webgl (l'utilitza el motor de joc que he fet servir) i websockets.

En el àmbit dels videojocs *online*, últimament s'està potenciant l'estil anomenat *pay to win*[1]. Això fa que molts jugadors vagin saltant de joc en joc, ja que cap acaba sent satisfactori pel jugador o no estan disposats a pagar per avançar en el joc.

D'altra banda, personalment els jocs que valen la pena són de quotes mensuals, això fa que jugadors no independents econòmicament no puguin jugar, o simplement alguns opinin que poden arribar a trobat el mateix o similar que sigui gratuït.

D'aquí va sorgit la idea de fer un videojoc que sigui gratuït amb compres integrades al joc, però que no afectin l'experiència dintre del joc. Conservant els valors dels jugadors de la vella escola. Fer un joc estimulant i addictiu, on tota mena de jugadors puguin gaudir amb un joc fet per un jugador per ells. Conservant totes les bones coses que he anat observant en tota mena de videojocs, als quals he jugat. I el més important fer-ho arribar al major número de persones al realitzar-ho per navegador web.

[1] Pay to win: joc on solen guanyar els jugadors que compren objectes o similars amb diners reals.

1.1. Propòsits i objectius

La principal motivació, per desenvolupar aquest projecte és la meva passió per el món web i els videojocs. Altres aspectes que m'han fet tirar endavant aquest projecte són les noves caracteristiques que ofereix el HTML5.

El propòsit principal d' aquest projecte és desenvolupar un joc PVP[2] en 2 dimensions i temps real per navegador. El joc serà per 2 jugadors en una mateixa partida, on tindran que lluitar fins que la vida del seu personatge baixi a 0.

1.2. Estructura de la documentació

Aquesta memòria estarà dividida en 14 capítols que considero els més importants. A continuació, els enumeraré i faré un breu resum del més important exposat a cada punt:

1. *Introducció, motivacions, propòsits i objectius del projecte:* S'explica una breu introducció al projecte, el perquè del seu desenvolupament i els objectius proposats.
2. *Estudi de la viabilitat:* Es justificaran els paràmetres necessaris que faran possible el desenvolupament d'aquest projecte. Entre ells estan els recursos d'infraestuctura, econòmics, humans, etc. I si el projecte és viable o no.
3. *Metodologia*: Conté la metodologia que farem servir per a l'evolució del sistema informàtic.
4. *Planificació:* S'exposarà l'estratègia seguida per assolir els objectius i la temporització.

[2] PVP: Terme fet servir per referir-se a jocs de jugador contra jugador on ambdós són humans.

5. *Marc de treball i conceptes previs:* Descriurem els conceptes previs que cal saber per poder entendre millor els capítols següents. També inclourem alguns passos duts a terme en l'aprenentatge del desenvolupament de videojocs.
6. *Requisits del sistema:* Es defineixen els requisits, funcionals i no funcionals, del programari. Aquí ajudarem a entendre els elements que envolten el sistema informàtic.
7. *Estudis i decisions:* Explicarem les possibilitats que tenim per al desenvolupament del programari, el software que hem fet servir i el perquè l'hem triat.
8. *Anàlisi i disseny del sistema:* Farem una anàlisi del problema que tenim entre mans i explicarem la solució a la qual hem arribat. Arribats a aquest punt exposarem el contingut dels punts anteriors d'una manera més formal, utilitzant l'enginyeria del software.
9. *Implementació i proves:* Explicarem el procés de desenvolupament, els problemes que ens hem trobat al llarg d'aquest i les solucions. També exposarem el model de classes i els mètodes més importants per la comprensió del programari desenvolupat.
10. *Implantació i resultats:* Mostrarem els resultats obtinguts del desenvolupament.
11. *Conclusions:* exposarem les conclusions extretes al finalitzar el projecte, les desviacions de la planificació inicial i els seus motius.
12. *Treball Futur:* Explicació del treball futur a realitzar per mantenir el programari actualitzat.
13. *Bibliografia:* Documentació consultada, digital i en paper, emprada per el desenvolupament del projecte.

2. Estudi de la viabilitat

Per desenvolupar aquest projecte, només ha estat necessari un ordinador portàtil amb un cost de 700 € i 3 mesos de feina per part meva. Suposem que hauria d'haver estat cobrant un sou, d'aproximadament 2.000 €/mes, comptem que és salari brut i tenint en compte la despesa d'autònoms. Això vol dir, que el cost del desenvolupament de l'*alpha*[3] ha costat 6.700 €. També hauríem de tenir en compte, les despeses d'aigua, llum, despeses vàries, uns 500 €/mes, així que parlem de 3.000 €/any. El software fet servir per desenvolupar el projecte, són de programari lliure, així que no hem invertit en res més que no hagi estat la maquinaria i el temps.

2.1. Recursos humans

En l'àmbit de recursos humans, per poder continuar desenvolupant el videojoc necessitarem diferents perfils. Un d'ells serà el compositor, un aspecte fonamental dels videojocs, és una bona banda sonora. Per altra banda, necessitarem dos dissenyadors, un del jocs i un altre pel web. El dissenyador de joc es dedicarà a crear els personatges i els elements del mapa. Per una altra banda, el dissenyador web s'encarregarà de què el web sigui usable i tingui un disseny amb consonància amb el que volem transmetre. Fins aquí tindríem l'equip de programació.

Els següent membres del equip s'incorporarien en cas d'assolir els objectius econòmics.

Una part important del equip és l'administrador de sistemes. Aquesta persona s'encarregaria de que en tot moment el sistema estigues en marxa, escalar el sistema, per tal, de fer créixer al mateix temps que augmentessin el nombre de persones

[3] Alpha: Fase del desenvolupament de software on encara és inestable, emprada per fer proves i solucionar errors. En el desenvolupament de videojocs sería on s'envien invitacions limitades, perquè la gent provi el joc i reporti errors.

connectades i altres tasques de manteniment rutinari (*backups*, comprovació dels *logs* d'error, etc.).

Arribats a aquest punt, ja tindríem el mínim d'equip tècnic que necessitaríem per poder continuar amb el desenvolupament del projecte. Ara hem de mirar la part de màrqueting, financera i legal.

A la part de màrqueting, hauríem de contractar una persona que en tot moment estigues estudiant el mercat, planificant la direcció cap a on ha d'anar el producte. Qui sería el creador de la campanya publicitària, triant els medis necessaris (digitals, en paper, etc.) per d'aquesta manera, poder arribar a fer-nos conèixer al públic. En el departament financer, tindrem una persona que s'encarregara de portar la comptabilitat de l'empresa i guiar les inversions de la mateixa, per tal de no acabar a la ruïna. En l'ambit legal, intentaríem que el perfil financer encaixes també en el legal, i d'aquesta manera ens estalviaríem una persona a l'inici. Que més tard contractaríem, quan creguem que l'empresa té les dimensions suficientement grans.

2.2. Viabilitat Econòmica

Econòmicament parlant, el projecte ha estat factible, gràcies a què realment no he cobrat cap sou i la maquinaria ha estat amortitzada en altres projectes que he realitzat.

Per recuperar la inversió inicial, i poder continuar tirant endavant, optarem per un model de negoci, amb micropagaments dins del joc, per obtenir elements. Ens basarem en un joc d'èxit que ja està assolit dintre del mercat internacional, el *League of Legends*.

Els jugadors podran comprar monedes en el joc, que després es podran gastar en aspectes del personatge, mesos de premium, mapes entre d'altres. El premium

consistirà en que si els servidors estan saturats, aquests tindran prioritat a l'hora d'entrar i podran crear mapes.

Després, vindrien servidors dedicats a grups de persones, que pagarien X diners al mes, per tal, de tenir un servidor només per ells, però sincronitzat amb la base de dades global.

Comptem que la mitja d'inversió per cada jugador, serà de 5€/ joc, per tant, per tenir els mateixos beneficis que pèrdues necessitarem uns 1400 jugadors (1352 jugadors exactament).

ACTIU		**PATRIMONI NET I PASSIU**	
ACTIU NO CORRENT		PN I PASSIU NO CORRENT	
IMMOB. INTANGIBLE (Servidor)	60	CAPITAL	3.005,06
IMMOB.TANGIBLE (Ordinador)	700	RESERVES	6.000
ACTIU CORRENT			
SOUS I SALARIS	6.000		
BANC	2.245,06		
TOTAL	**9.005,06**	**TOTAL**	**9.005,06**

2.3. Viabilitat Tecnològica

Respecte a la viabilitat tecnològica, he de dir que per desenvolupar el projecte no he necessitat una gran infraestructura. L'ordinador utilitzat, és suficient potent, per fer córrer la base de dades, el servidor i el client web.

Per fer algunes proves, vam provar un *cloud web* anomenat *nodejitsu*, amb un cost de 20 € al mes. El vam fer servir durant 3 mesos, pel qual vam invertir 60 € més. Per alguns problemes amb el servei, el vam acabant donant de baixa i vam centralitzar-ho tot en l'ordinador en qüestió.

3. Metodologia

Una metodologia en desenvolupament de *software* és una combinació de processos genèrics, defineix amb precisió els artefactes, rols, activitats involucrats, les practiques i també tècniques recomanades. En aquest projecte hem utilitzat un procés iteratiu.

S'ha decidit aquest procés per tal de seguir una pauta on sempre tenim 2 passos per cada etapa. El primer pas consisteix en dur a terme la tasca proposada, i en el segon farem les proves del treball realitzat en el pas 1. Un cop hem finalitzat les proves, si no són satisfactòries tornem al primer pas.

Com és normal en un procés iteratiu fins que no acabem un punt no passem al següent. Els passos seguits en el desenvolupament del nostre projecte són:

1. Determinació i estudi dels requisits.
2. Repassem els requisits, si tot és correcte, si creiem que encara no són correctes o estan inacabats tornem al pas u, sinó avancem al pas següent.
3. Presa de decisions tècniques, llenguatge de programació i eines que utilitzarem.
4. Disseny del sistema, tenint en compte les decisions preses en els passos anteriors.
5. Comprovem que el sistema dissenyat englobi tots els requisits, si compleix amb els objectius avancem al pas següent, sinó tornem al pas anterior.
6. Desenvolupament del disseny realitzat en el pas anterior.
7. Si veiem que durant el desenvolupament necessitem modificar requisits o parts del disseny proposat en el pas 4 perquè s'ajustin més a la realitat, tornem al punt 4.
8. Testeig del desenvolupament, aquí tindríem el procés de *QA*[4]*,* si en aquest punt no hem complert tots els requeriments definits i no passem el procés de *QA,*

[4] QA: *quality assurance*, assegurar la qualitat del treball desenvolupat.

tornem al pas anterior per solucionar *bugs*[5] i tornar a programar algun requisit que no complíssim.

9. Fer la documentació del projecte.
10. Revisió de la documentació, si tenim errors, tornem al pas anterior per fer les correccions necessàries, sinó donem per tancat aquesta fase del projecte.

A continuació exposem aquest procés iteratiu en el següent diagrama de activitat:

[5] Bugs: errors en el programari.

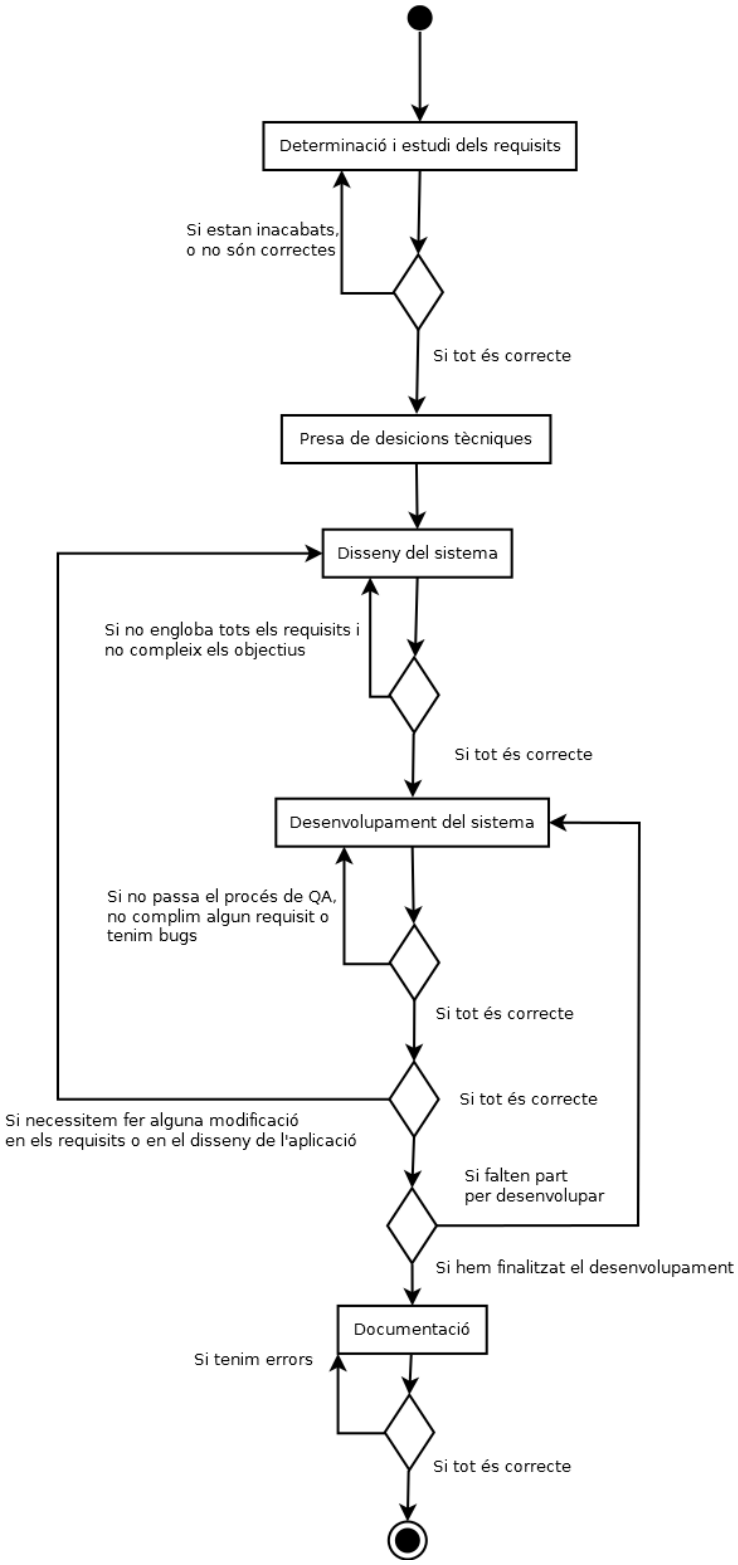

Figura 1. Diagrama d'activitat de la metodologia

4. Planificació

El projecte es va planificar al setembre del 2014, però no és va començar a desenvolupar fins al desembre del 2014 per motius de feina. La data proposada d'entrega va ser a la convocatòria de juny del 2015. Es va decidir que es faria el projecte en un període de 6 mesos.

La primera fase, va ser un procés de *brainstorming*[6] per determinar els requisits del joc. Quan es va acabar el *brainstorming,* es va fer un estudi de quins requisits s'implementarien en aquesta primera fase i quins no.

A causa de les festivitats de nadal es va fer una pausa per descansar i assimilar les decisions de la primera fase.

Tot seguit comença la segona fase, la del disseny. Aquesta fase va consistir en determinar el *bounded context*[7] del nostre projecte. Un cop es va delimitar el problema, es va prosseguir amb la identificació dels actors i quins casos d'ús tindria cadascun.

Identificats els casos d'ús, ja nomes queden 2 punts importants en el disseny de l'aplicació, el model de dades i l'estructura de mòduls (en javascript no tenim classes com a tal).

En el primer punt es va prendre la decisió, de quin tipus de base de dades faríem servir (relacion o no relacional), un cop presa la decisió, es va procedir a la creació de l'esquema de dades.

[6] Brainstorming: recollida d'idees
[7] Bounded context: Delimitació del problema.

En el segon punt, es va fer el disseny de quins mòduls crearíem per l'aplicació. Mòduls per encapsular les funcions que es crien en l'encaminador, funcions de *sockets*, funcions d'utilitats, la part de client, etc.

Un cop acabat el disseny de l'aplicació, es va passar a la fase de desenvolupament. Aquesta va consistir en plasmar tot el que havíem pensat i decidit en les fases anterior a codi. Aquí es va veure també les mancances del disseny i es va poder rectificar, perquè s'adaptes millor a la realitat.

En acabar la implementació, ja nomes quedava millorar la maquetació de la web, per millorar-la visualment i a nivell de navegació (encara que no tenim gaires seccions).

L'ultima fase, consisteix en fer la redacció final de la documentació recollida en aquesta memòria. És a dir, agafar totes les idees i experiències viscudes durant el procés d' estudi, disseny i desenvolupament del projecte i plasmar-ho.

Per veure millor com s'han dut a terme aquestes fases en una linea temporal, s'adjunta un diagrama de gantt, aquest s'ha fet amb l'eina *teamgantt,* que ofereix una prova gratuïta amb funcionalitats bàsiques de 30 dies.

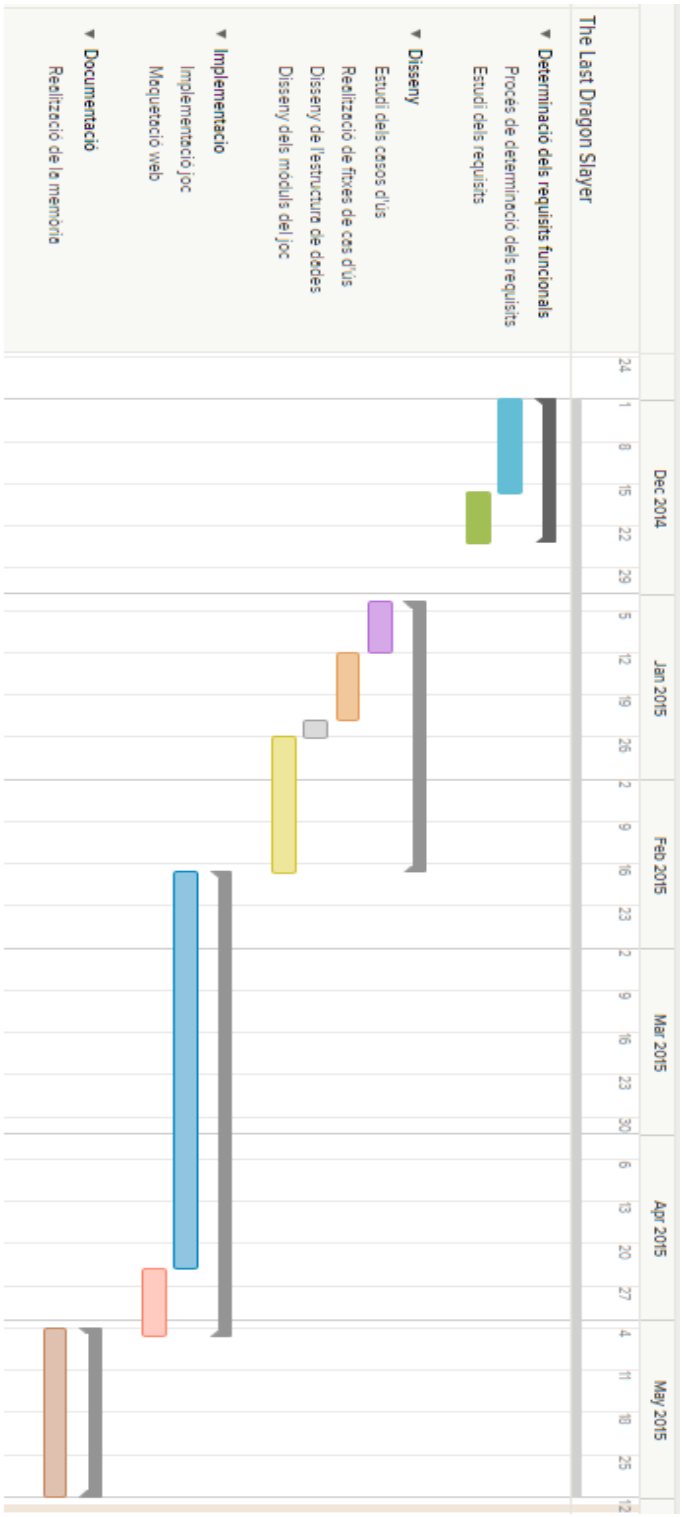

Figura 2. Diagrama de Gantt

5. Marc de treball i conceptes previs

En aquest apartat, posarem al lector en el marc de treball, explicant els conceptes previs que necessitarem per entendre les decisions preses i entendre millor el disseny del projecte.

5.1. Arquitectura client-servidor

L'arquitectura client-servidor és una distribuïda entre diversos processadors, els clients que sol·liciten uns serveis i els servidors els proporcionen.

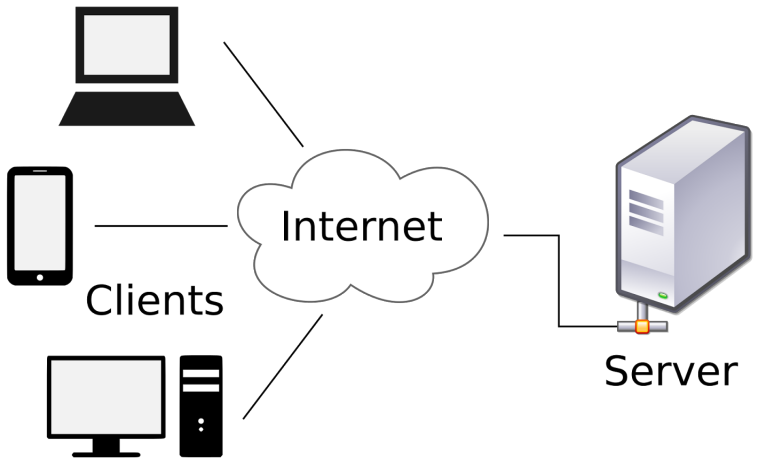

Figura 3. Exemple Arquitectura Client-Servidor[8]

El client és com un *message dispatcher*[9], envia un missatge en un protocol especific (en el nostre cas HTTP i en el cas del sockets TCP). El servidor és com un *listener*[10], està escoltant peticions i quan les rep descodifica el missatge, el processa i envia una resposta.

[8] Model client servidor - http://es.wikipedia.org/wiki/Cliente-servidor#/media/File:Client-server-model.svg
[9] Message dispatcher: enviador de missatges
[10] Listener: Oient, en el nostre cas escolta peticions HTTP i TCP.

5.2. Aplicacions web

Una aplicació web és un *software* que s'executa sobre un servidor web. El client és principalment un navegador web (per exemple mozilla firefox, google chrome, entre d'altres).

Normalment es divideix en 2 parts: el *frontend* i el *backend*. La part de *backend* és on consultem totes les dades i on apliquem a lògica de negoci. La part de *frontend* és la interfície gràfica.

La part gràfica es basa en un llenguatge anomenat HTML per donar estructura al document (anomenat DOM[11]). Per manipular aquest DOM i fer crides al servidor per ampliar les dades que estem mostrant utilitzem un llenguatge de navegador anomenat *Javascript*.

5.3. Programació amb javascript

Javascript és un llenguatge de programació interpretat, funcional, imperatiu i de tipatge feble.

En la majoria de llenguatges de programació totes les operacions són seqüencials, és a dir, el codi té un fil que es va seguint de forma contínua. Mentre que en javascript la programació és asíncrona. Per exemple, podem enviar una petició al servidor per AJAX i no fa falta què esperem a que respongui, li establim unes funcions que s'executaran quan rebem la resposta però si tenim més codi a continuació segueix executant.

[11] DOM: Document object model, arbre de nodes que construeix el document HTML.

Això s'aconsegueix treballant amb un *event loop* [17]*,* aquest model es compon de 3 elements: *heap*, *stack* i *queue.*

Encara que es defineix com un llenguatge orientat a objectes, realment treballem amb pseudo-classes, aquestes classes es basen amb prototips. Els objectes es creen a partir d'una funció constructora on aquesta té assignada un conjunt d'atributs i funcions al seu prototip (prototype).

Per acabar cal remarcar que el javascript està pensat per treballar amb el DOM, però amb els últims avenços han aconseguit fer servir el motor de google chrome per executar javascript al servidor.

5.4. Web Sockets

Els web sockets serveixen per crear un canal de comunicació contínua entre un client web i un servidor.

Per fer la connexió amb el servidor, primer s'envia una petició HTTP i quan el servidor contesta, llavors es crea el canal persistent. D'aquesta manera podem enviar events de forma contínua.

Cal aclarir que encara no es pot arribar a establir connexió entre dos clients web a partir de web socket, sempre hem de tenir un servidor que faci d'intermediari i els comuniqui.

5.5. Canvas

El canvas és un nou element arribat amb el HTML5 que ens permet dibuixar gràfics en un navegador web. Té dos atributs obligatoris l'amplada i l'alçada.

Normalment dibuixem en ell via javascript, actualment es poden dibuixar tan gràfics 2D com 3D (amb *WebGL*).

5.6. Elements i funcionament dels jocs

En aquest apartat, explicarem els conceptes bàsics que es necessiten saber per entendre com funciona un joc a nivell intern.

5.6.1. Atles de textures

Un atles de textures és un conjunt de *tiles* agrupats. Un *tile* és un quadrat, en el nostre cas de 32x32px, que representa un element que s'ha de dibuixar. Un podria ser el personatge mirant cap dalt, un tros de mapa qualsevol element del joc.

Figura 4. Exemple d'atles de personatge

Els atles no deixen de ser imatges, i necessitem que pesin el menys possible perquè el servidor les envii ràpidament. Per això es recomanable comprimir aquestes amb un *texture packer*. Això agrupara tots els *tiles* el més comprimit possible i ens generara un fitxer amb les coordenades de cada element. Que nosaltres llegirem a posteriori per dibuixar cada element i fer les animacions.

D'altra banda també necessitem tenir el menor número d'imatges possible per minimitzar les peticions (cosa que fa que augmenti el temps de càrrega).

Un atles es compon del conjunt d'imatges que venen representades en un JSON. Aquest JSON ens donara la representació de la imatge perquè el motor pugi dibuixar aquests elements dintre del joc. Com podem veure millor a ***Figura 6*** veiem que cada tile[12] té la informació per poder-lo dibuixar al nostre *canvas*.

```
"walking_down_1": {
        "frame": {"x":246, "y":190, "w":28, "h":50},
        "spriteSourceSize": {"x":0,"y":0,"w":28,"h":50},
        "sourceSize": {"w":28,"h":50}
},
"walking_down_2": {
        "frame": {"x":275, "y":241, "w":28, "h":50},
        "spriteSourceSize": {"x":0,"y":0,"w":28,"h":50},
        "sourceSize": {"w":28,"h":50}
},
"walking_down_3": {
        "frame": {"x":276, "y":103, "w":27, "h":51},
        "spriteSourceSize": {"x":0,"y":0,"w":27,"h":51},
        "sourceSize": {"w":27,"h":51}
},
"walking_down_4": {
        "frame": {"x":304, "y":153, "w":26, "h":50},
        "spriteSourceSize": {"x":0,"y":0,"w":26,"h":50},
        "sourceSize": {"w":26,"h":50}
},
```

Figura 5. Fragment del atles de personatge

[12] Tile: Quadrat dintre del atles que representa un element a dibuixar.

5.6.2. Main loop

El procés principal de un joc és un bucle que s'executa tantes vegades cada segon con *frames* volem tenir (60 *frames* per segon 60 cops de bucle per cada segon).

El main loop és composa de 3 fases:

- Inicialització: Carrega els elements multimèdia (imatges, sons, etc.), inicialitza mapa, personatges i sistema de físiques. En definitiva tot el necessari per poder continuar amb els estats de actualització i dibuixar
- Actualització: Prepara tots els objectes per ser dibuixats. Per exemple, detecta col·lisions, baixa la vida dels personatges, processa els inputs del usuari, etc.
- Dibuixar: Agafa tota la informació que hem tractat en l'actualització i la dibuixa, en el nostre cas en el *canvas*.

5.7. Bases de dades no relacionals

Les bases de dades no relacionals són bases de dades que difereixen amb el model de les tradicionals.

No tenen esquema (en una mateixa col·lecció podríem guardar diferents tipus de objectes amb diferents atributs), no permeten fer joins entre taules. No hi ha relacions entre taules (no tenim claus foranes).

Hi ha varies categories d'aquest tipus de bases de dades, les de clau valor (un exemple seria el redis [18]), orientades a columnes (un exemple seria Cassandra [19]), orientada a documents (un exemple seria mongodb [20], el que hem fet servir per aquest projecte) i orientada a grafs (un exemple seria Neo4j [21]).

Les avantatges que ofereix sobre les bases de dades SQL entre d'altres és que permet treballar amb un flux més gran de dades, però ens penalitza ja que no ens assegura el conjunt de característiques ACID[13].

5.8. Sistema de control de versions

Un sistema de control de versions registra en cada moment els canvis d'un o un conjunt de fitxer, al llarg del temps. D'aquesta manera podem recuperar un estat anterior en qualsevol moment. El més important d'aquests sistemes és el poder tirar un producte a un estat anterior que és estable si alguna cosa no ha anat bé del tot amb l'estat actual.

Quan es treballa en equip és molt important tenir un control de versions, per tal de coordinar tot el codi en repositoris. D'aquesta manera s'agilitza tot el procés de pujada del producte i també et dóna la seguretat de que el codi està en un repositori. Alguns exemples de sistemes de control de versions són: git, subversion i mercurial.

[13] ACID: consistencia, aïllament i durabilitat

6. Requisits del sistema

Després de fer l'estudi dels requisits, els separem en 2 tipus: funcionals i no funcionals. Els requisits funcionals són els serveis que ha de proporcionar el nostre sistema. Per altra banda, els requisits no funcionals són les característiques que d'una forma o una altra poden limitar el sistema.

6.1. Requisits no funcionals

1. Disposar de connexió a internet.
2. Disposar d'un navegador web compatible amb html5 i canvas.

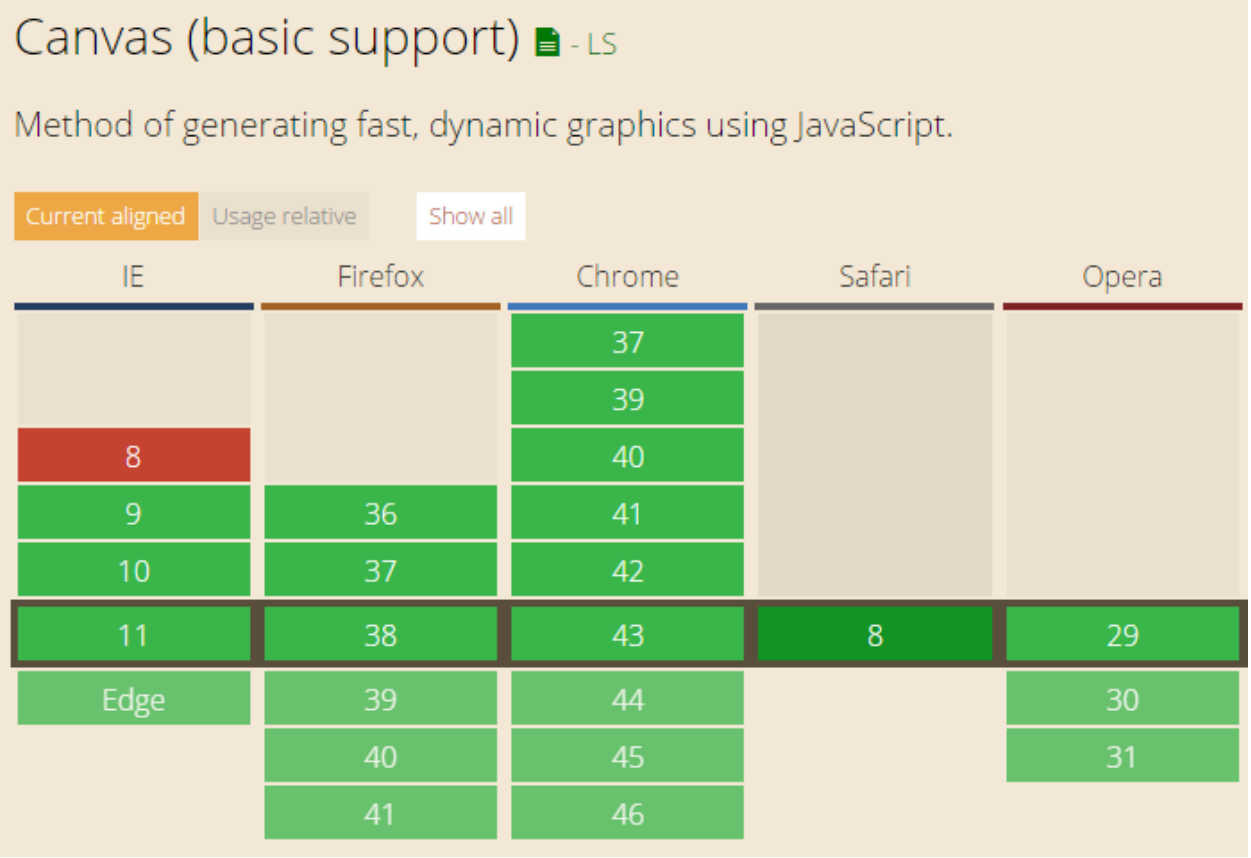

Figura 6. Compatibilitat de navegadors amb *canvas*[14]

[14] Dades extretes de: http://caniuse.com/#feat=canvas

Els navegadors ressaltats, tot i que la compatibilitat és 100%, nosaltres l'hem fet per google chrome, són els òptims per poder fer córrer el joc sense problemes.

3. Per la correcta visualització de la web recomanem els navegadors ressaltats anteriors també, ja que utilitzem la llibreria Bootstrap per al disseny visual de la web i han de ser compatibles amb CSS3.
4. Pel correcte funcionament del joc caldrà un ordinador amb 2 GB de ram com a mínim i un processador *intel pentium 4.* Aquests són els requeriments que faran que el joc vagi fluid.
5. Configuració i posada en marxa del sistema al servidor.

6.2. Requisits funcionals

En aquest punt, partirem de la base que tenim dos tipus d'usuaris. L'**usuari anònim** i el **"jugador"**(usuari registrat).

Un usuari anònim ha de poder fer 3 accions bàsiques:

1. Veure el rànquing, perquè està obert a tots els usuaris.
2. Registrar-se, és a dir, convertir-se en un usuari "jugador".
3. Accedir al joc.

D'altra banda, l'usuari "jugador" que és un usuari anònim, però que ja ha accedit a l'aplicació, ha de poder:

1. Unir-se a una partida
2. Crear una partida
3. Un cop dins d'una partida, jugar i xatejar amb el jugador rival.
4. Pot veure el rànquing
5. Pot sortir de l'aplicació, passar a ser un jugador anònim.

Un cop explicada aquesta part, passem a explicar els 2 actors no humans. El **servidor** i el ***timer* (temps).**

El servidor en tot moment ha d'estar connectat per poder fer les següents accions:

1. Rebre peticions HTTP i enviar les respostes als clients.
2. Rebre peticions TCP i enviar les respostes als clients.
3. Gestionar les dades de la base de dades.

El *timer,* diguéssim que són les accions que s'executen en el *main loop* explicat en l'apartat de conceptes previs. S'encarrega de:

1. Empaquetar i enviar les dades a través del *socket* al servidor.
2. Refrescar el *canvas*. Aquesta funció inclou rebre el relatiu al personatge remot, processar aquesta informació i actualitzar-lo.
3. Calcular les col·lisions.

7. Estudis i decisions

En aquest apartat explicarem les tecnologies emprades en el desenvolupament i perquè les hem escollit.

7.1. Client

En la part de client, els llenguatges emprats són html5+css3 per la maquetació i l'estructura de la web, i el JavaScript per aplicar la lògica de l'aplicació. Ara han sortit un seguit de llenguatges que suporten els navegadors com el *CoffeScript*, *TypeScript*, etc. No hem fet servir cap d'aquests, perquè com tots compilen sobre javascript pensem que no ens aportarien res de nou.

Hem separat en dos punts les decisions importants referents al client. El primer punt parlarem de la decisió presa pel motor del joc, i en el segon punt dels *frameworks* i llibreries utilitzades per desenvolupar la part del client.

7.1.1. Motor del joc

Figura 7. Logotips de motors html5

A l'hora de desenvolupar el joc, una de les parts importants és triar un motor potent i fàcil de fer servir. A continuació, exposarem cinc dels motors que més ens van agradar i exposarem perquè finalment el utilitzat va ser el Phaserjs.

Abans d'exposar perquè ens vam decantar per el Phaserjs, tenim que recapitular una mica. Per pendre aquesta decisió hem de tenir en compte els elements que tenim en el joc. En el nostre cas són mapes i personatges.

Els mapes i els personatges són imatges que és representen a traves de un JSON on és guarden les animacions. Per construïr aquests JSON's hem fet servir 2 programes.

Pel mapa hem fet servir el **tiled map editor** [22]. Aquest editor de mapes és programari lliure, molt fàcil de fer servir. L'hem triat perquè a part de donar-nos la opció de crear capes de *tiles,* també ens dóna l'opció de crear capes d'objectes (que ens serviran per les col·lisions per exemple).

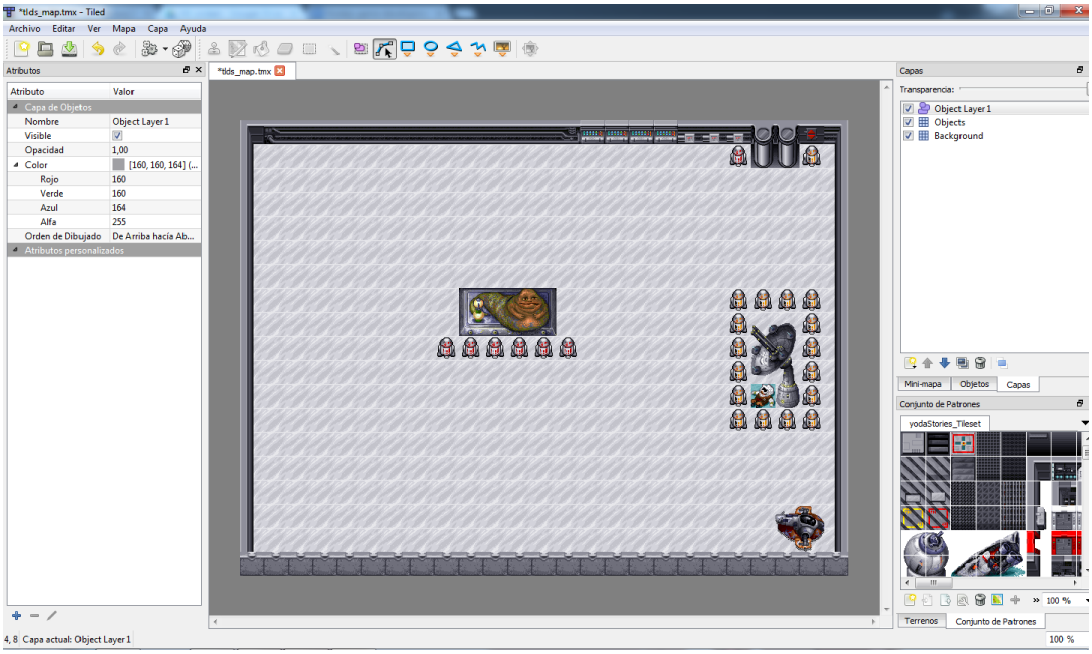

Figura 8. Tiled map editor

En canvi pel personatge, hem fet servir un texture packer de programari lliure també, el **shoebox** [23]**.** Després d'estar buscant un texture packer gratuït, aquest ha sigut l'únic que vaig trobar que garantís tot el que necessitàvem per crear el atles.

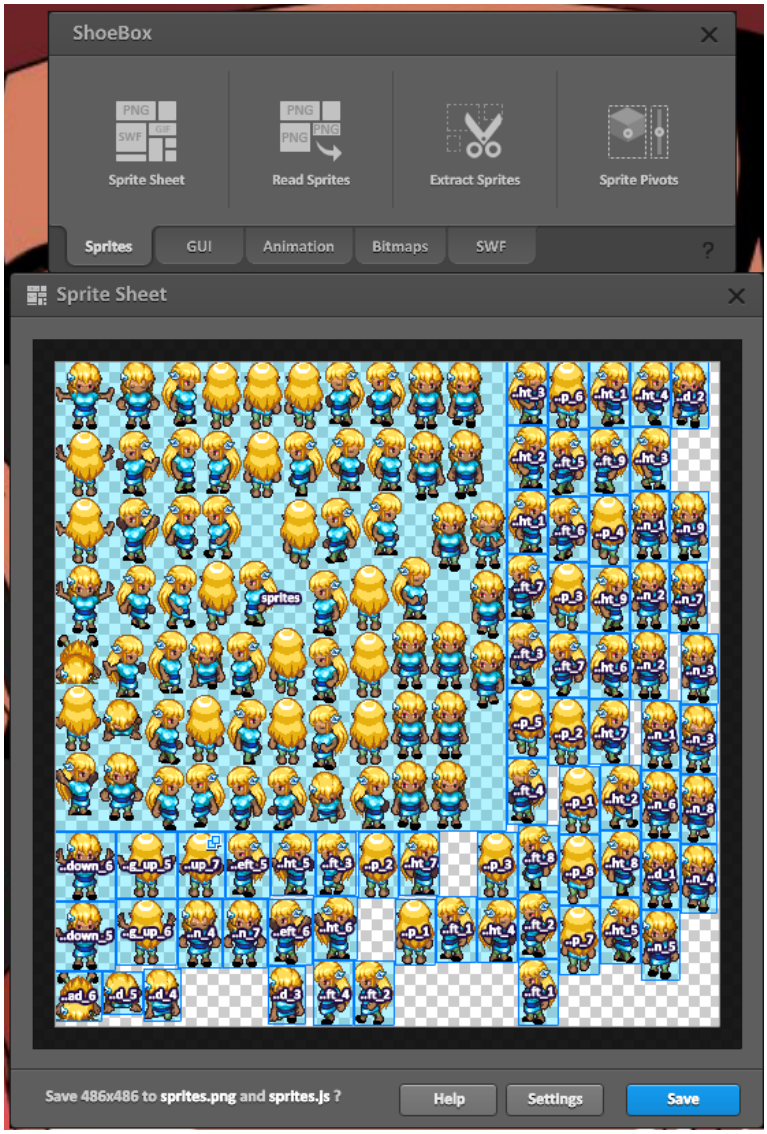

Figura 9. Shoebox

Un cop vam tenir aquests atles necessitàvem algun motor que pogués treballar bé amb aquest sistema. Per això vam començar mirant el **Impactjs** [24] un motor molt popular, però que vam descartar ràpid perquè no és de programari lliure.

El segon motor que vam estar investigant va ser el **kiwijs** [25], donava moltes facilitats per implementar el joc, podia treballar amb el nostre sistema perfectament,

però no portava integrat un sistema de físiques per aquest motiu va quedar descartat també.

El **melonjs** [26] és un motor que encara està en desenvolupament i encara que ens va donar la sensació de què era molt potent, per precaució, el vam descartar també.

Per acabar amb els descartats parlem del **gamejs** [27]**.** Aquest motor està basat amb una llibreria de ***python*** per fer videojocs. En investigar una mica ens vam adonar que comparat amb el **phaserjs** no ens acabava de convèncer l'estructura que utilitzaven en l'àmbit de codi per programar el joc. Així que aquest també va quedar descartat.

El motor que vam triar ha estat el **phaserjs** [28]**,** aquest motor és compatible amb *canvas* i *webgl*. També porta integrats variïs motors de físiques, cosa que el fa molt interessant.

Els avantatges d'utilitzar **phaserjs** respecte els altres són:

1. Té integrat variïs motors de físiques.
2. És de programari lliure.
3. Comunitat molt amplia, ja que a més **photonstorm,** l'empresa creadora, coopera amb la llibreria **pixi.js** [29] també.
4. Compatible amb la majoria de navegadors moderns i també dispositius mòbils, encara que el nostre joc no està adaptat per treballar amb aquests últims.
5. Bona estructuració de codi, permet fer herència de les seves classes.
6. Podem crear variïs estats del joc. Cadascun amb la seva càrrega de recursos (imatges, sons, ...). I el seu propi **main loop**.

7. Com suporta **webgl,** si el navegador és compatible, millora moltíssim el rendiment del joc.
8. Sí necessitem més utilitats pel motor, és extensible, et permet crear els teus propis plugins per al joc.

7.1.2. Llibreries utilitzades al client

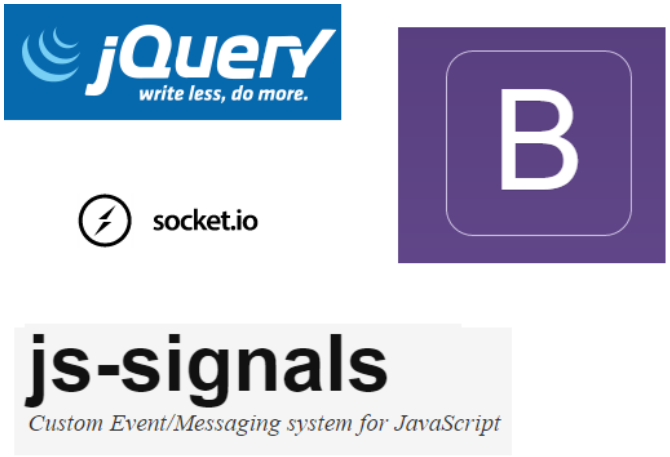

Figura 10. Logotips de llibreries i *frameworks* de la part del client.

A la part client, es va prendre la decisió de fer servir llibreries estàndards, que fa servir la gran majoria de persones que programen per entorn web.

El **jQuery** [1] és un framework que ens dóna una capa per treballar amb el *DOM.* Les avantatges de treballar amb ell són moltes, una de les més importants per mi és la gran potencia a l'hora d'agafar elements del DOM a través dels selectors, ja siguin CSS o alguns personalitzats de la pròpia llibreria. D'altra banda, també té molt suport de la comunitat, molta documentació i és codi lliure.

Després per complementar les funcions de jQuery, és va optar per una llibreria per tractar *events*, **signals-js** [6]. Un *signal* és un disparador d'*events* que conté un *array* de *listeners*[15]. Sobretot és va decidir aquesta, per la comoditat de crear *events* personalitzats com a objectes i afegir de forma fàcil els listeners. També cal recordar que aquesta eina és de codi lliure i al encapsular una funció també té més compatiblitat entre navegadors.

A la part de sockets ens vam decantar per **socket.io** [5], hi havia altres llibreries com el *now.js,* però estaven pitjor documentades i eren més complicades de fer servir. També cal aclarir que el principal motiu que ens va fer escollir aquesta llibreria va ser que tenia versió tant per client, com per servidor. D'aquesta manera ens evitàvem qualsevol possible incompatibilitat de paquets entre llibreries.

Fins aquí ja tenim totes les llibreries que farem servir per a la lògica de l'aplicació.

Per la maquetació i el disseny web s'ha utilitzat la llibreria **Bootstrap** [2]. Aquesta llibreria és un conjunt de CSS i javascript que ens permeten fer una maquetació ràpida i amb facilitat. S'ha optat per fer servir aquesta llibreria en comptes de CSS+HTML només.

El principal motiu ha estat la facilitat a l'hora de maquetar qualsevol cosa i l'estructura que proporciona. A més facilita un disseny *responsive*[16], el qual permet fer arribar la nostra maquetació a més dispositius.

[15] *Listener*: Procés que quan escolta l'event executa una funció.
[16] Responsive: Que s'adapta a qualsevol dispositiu.

7.2. Servidor

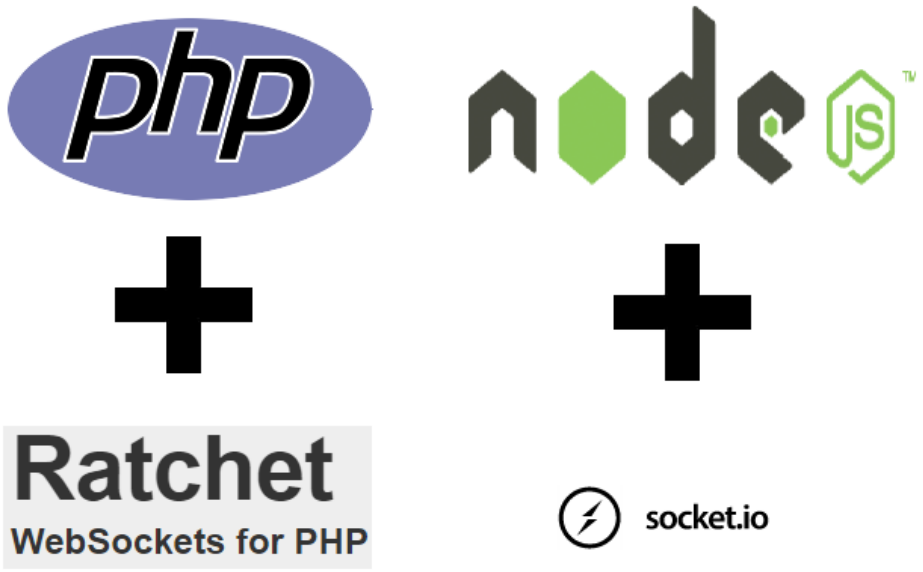

Figura 11. Logotips llenguatge servidor i llibreria de *sockets*.

A l'hora de triar un llenguatge pel servidor, vam estar entre **nodejs** amb la llibreria ***socket.io*** per gestionar els sockets. I després amb ***PHP*** i ***Ratchet***, l'equivalent a socket.io en *PHP*.

La decisió final va ser utilitzar *Nodejs*. D'aquesta manera també unificàvem el llenguatge de programació a un de sól. Comparativa sobre PHP i Nodejs [30].

Avantatges sobre *PHP*:

1. Pot processar més peticions.
2. El percentatge de peticions fallides és més baix.
3. Velocitat de transferència més elevada.
4. Per rebre peticions *HTTP*, *PHP* necessita d'un servidor *HTTP* extern com seria *Apache* o *Nginx*. *Nodejs* s'executa i la mateixa instància ja crea el servidor.

Desavantatges sobre *PHP*:

1. Tecnologia més jove.
2. Comunitat menys amplia que la de *PHP*.

Per nosaltres el avantatge més gran d'utilitzar *socket.io* sobre *rachet*, va ser el no haver d' aprendre a utilitzar una llibreria per al client i una per al servidor. Ja que, aquesta en té per ambdues bandes.

Amb node vam integrar un framework anomenat **expressjs** [31]**.** Aquest va ser escollit respecte a d'altres com **flatiron,** perquè era el que més exemples tenia i millor documentació.

I com veurem en el punt següent Nodejs va ésser escollit per la seva bona integració amb la base de dades utilitzada en el nostre projecte, **Mongodb**.

7.2.1. Base de dades

Figura 12. Logotips bases de dades

La decisió de quina base de dades faríem servir va ser la més difícil de totes. Després de revisar documentació de diferents tipus de bases de dades, vam limitar la llista a dos, mongodb i mysql.

L'opció fàcil hauria estat fer la base de dades amb mysql, ja que l'hem après a la universitat i he estat treballant amb aquest tipus.

Però ens vam acabar decantant pel **mongodb** [20]**.**

Mongodb és una base de dades no relacional, orientada a documents. No té esquema de dades i com que no té relacions, no permet fer *joins* entre col·leccions. Llavors la pregunta és: **per què Mongodb?**

En aquesta base de dades el format en el qual es guarden els documents és JSON[17]. Javascript treballa amb JSON's de forma nativa. Per tant quan rep les dades no ha de fer cap tipus de transformació, ja les té en el format que ell necessita.

Com és *schemaless* (sense esquema), ens dóna molta més flexibilitat a l'hora de guardar dades. No fa falta que tinguem columnes en blanc o *null*, simplement no les afegim a l'objecte que anem a inserir i així estalviem espai. Això és una eina de dos talls, ja que ens proporciona flexibilitat, però pot ser caòtic sí no tenim cura de què estem guardant.

Arribats a aquest punt nosaltres sí que necessitàvem un esquema mínim. D'aquí que utilitzéssim **moongose** [32]. *Moongose* és un ODM (*Object Document Mapper*) per

[17] JSON: Javascript object notation.

5mongodb. Ens permet crear models, que després es traduiran amb objectes que s'inseriran a la base de dades.

7.3. Entorn de desenvolupament

Figura 13. Entorn de desenvolupament

L'entorn de desenvolupament per programar ha estat l'editor de text **sublime text** [33]**.** No hem utilitzar cap IDE[18], ja que en el sistema operatiu que fem servir la gran majoria d' aquests consumeixen molta memòria. Sublime text és un editor de text

[18] IDE: Entorn de desenvolupament integrat.

lleuger i extensible mitjançant *plugins*. Nosaltres hem fet servir el **JSLint**, que servia per detectar errors de sintaxi a l'hora de programar.

Respecte al sistema de control de versions ens hem decantat per utilitzar **GIT** [34]. Git ens proporciona un sistema distribuït, ràpid i eficient. La corba d'aprenentatge és més lenta respecte altres sistemes de versions, perquè és un sistema amb moltes comandes i molts conceptes.

Pel que fa a un dels seus competidors, com ara el **subversion** [35], els avantatges són:

1. Treballar amb branques és molt més senzill.
2. Fer el *merge* del codi de les diferents branques és més senzill i menys critic que amb *svn*.
3. Si perds la connexió, pots seguir fent *commits* i així sempre tens la possibilitat de tornar a un estat anterior més fàcilment, ja que hi ha un pas intermedi amb un servidor local. Amb *svn* si no tens connexió amb el servidor, no pots pujar el codi.
4. Per projectes amb molts contribuïdors (milers) és molt més escalable.
5. Ocupa molt menys espai a disc.
6. Repositoris gratuïts per *open-source* coneguts, un exemple serien bitbucket i github.

Com a contrapartida, cal dir que *svn* té una eina molt intuïtiva i fàcil de fer servir, ja que s'integra amb la interfície del sistema operatiu. I personalment que opino que això li falta al GIT, una bona eina visual com el TortoiseSVN [36].

Per acabar amb l'entorn de desenvolupament que hem fet servir, parlarem del repositori de GIT remot.

Hem escollit **github**, no tenim cap raó en especial. Si l'hem utilitzat en comptes del **bitbucket** és, perquè regalaven algunes característiques de pagament si ets estudiant.

8. Anàlisi i disseny del sistema

A continuació detallarem el disseny proposat per solucionar el nostre problema i a partir de diagrames de cas d'us i les seves respectives fitxes exposarem la solució proposada. Aquí també exposarem el disseny de classes emprat, col·leccions de dades i el flux de missatges dels sockets.

8.1. Diagrames de cas d'ús

En el nostre sistema s'ha proposat la figura de 4 actors:

- Usuari anònim
- Jugador registrat
- Servidor
- *Timer*

A partir d'aquest 4 exposarem els seus casos d'ús.

8.1.1. Usuari anònim

En aquest apartat exposarem els casos d'ús que pot realitzar un usuari anònim, així com les seves fitxes de cas d'ús per veure en profunditat quines accions pot fer i com funcionen internament.

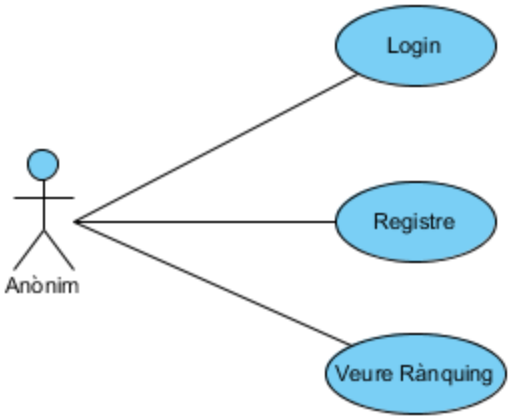

Figura 14. Casos d'ús usuari anònim.

Cas d'us Registre	
Actor:	Anònim
Versió:	1.0
Descripció:	Registrem un jugador a la base de dades i fa el login.
Pre condició:	-
Flux principal:	1 - Introduir àlies i contrasenya 2 - Si les dades són correctes 2.1 - Guardem les dades a base de dades. 2.2 - Procedim al pas 2 del cas d'us login.
Flux alternatiu:	2 - Si l'alias ja existeix 2.1 - Es mostra l'error per pantalla
Post condició:	

Cas d'us *Login*	
Actor:	Anònim
Versió:	1.0
Descripció:	Dóna accès als usuaris.
Pre condició:	L'usuari està registrat
Flux principal:	1 - Introduir àlies i contrasenya. 2 - Si les dades són correctes. 2.1 - Guardem les dades en sessió. 2.2 - Redirecció al joc.
Flux alternatiu:	2 - Si la contrasenya o l' àlies no són correctes. 2.1 - Es mostra l'error per pantalla.
Post condició:	L'usuari accedeix a la part privada.

Cas d'us Veure Rànquing	
Actor:	Anònim
Versió:	1.0
Descripció:	Es mostra un llistat amb els 10 jugadors amb més victòries.
Pre condició:	Algun jugador ha guanyat una partida.
Flux principal:	1 - Cliquem al menú. 2 - Es mostren els àlies dels jugadors i el número de victòries.
Flux alternatiu:	-
Post condició:	Rànquing visible.

8.1.2. Jugador registrat

En aquest apartat, exposarem els casos d'ús que pot realitzar un jugador registrat, així com les seves fitxes de cas d'ús, per veure en profunditat quines accions pot fer i com funcionen internament.

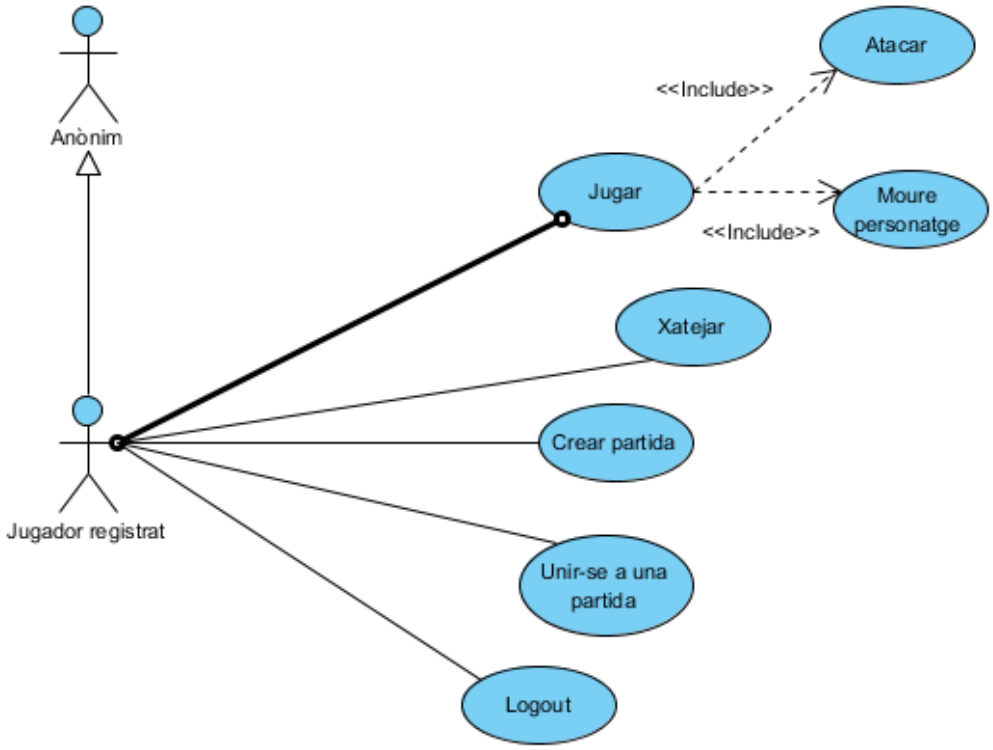

Figura 15. casos d'ús del jugador registrat

Cas d'us Crear partida	
Actor:	Registrat
Versió:	1.0
Descripció:	Crea una sala on es podrà unir un altre usuari.
Pre condició:	L'usuari no està a cap sala.
Flux principal:	1 - Posa nom a la sala 2 - El *socket* es connecta a la sala en concret. 3 - Es carrega l'estat inicial del joc. 4 - Es manté a l'espera de connexió d'un altre usuari.
Flux alternatiu:	-
Post condició:	La sala queda creada i el joc queda inicialitzat.

Cas d'us Unir-se a una partida	
Actor:	Registrat
Versió:	1.0
Descripció:	El jugador s'uneix a una sala i comença la partida.
Pre condició:	Hi ha 1 o més sales creades i no plenes (número de jugadors < 2)
Flux principal:	1 - L'usuari tria una sala de la llista. 2 - El *socket* es connecta a la sala que ha triat. 3 - Es carrega l'estat inicial del joc. 4 - Sincronitza el jugador remot.
Flux alternatiu:	2 - Si la sala esta plena. 2.1 - Es mostra missatge d'error.
Post condició:	La partida ja està preparada per començar a jugar.

Cas d'us Xatejar	
Actor:	Registrat
Versió:	1.0
Descripció:	Intercanvi de missatges entre dos usuaris en una mateixa sala.
Pre condició:	El jugador s'ha unit a una sala.
Flux principal:	1 - Si enviem missatge. 1.1 - Emplenem el *input*. 1.2 - Cliquem el botó d'enviar. 1.3 - El missatge s'envia a través del *socket*. 1.4 - El missatge s'encadena al llistat de missatges. 2 - Si rebem missatge. 2.1 - El missatge s'encadena al llistat de missatges.
Flux alternatiu:	-
Post condició:	Tenim més missatges al llistat.

Cas d'us Logout	
Actor:	Registrat
Versió:	1.0
Descripció:	Passem de ser jugador registrat a usuari anònim.
Precondició:	Hem accedit a l'aplicació a través del login.
Flux principal:	1 - Cliquem al botó de logout. 2 - S'esborra la sessió. 3 - Ens redirigeix al login.
Flux alternatiu:	-
Postcondició:	Som un usuari anònim.

Cas d'us Jugar	
Actor:	Registrat
Versió:	1.0
Descripció:	Flux principal del joc.
Precondició:	Han entrat 2 usuaris a una sala.
Flux principal:	1 - Càrrega d'arxius multimèdia (imatges, sons, ...) 2 - Càrrega de mapa. 3 - Mentre els dos jugadors estiguin vius (vida > 0) 3.1 - Si el jugador polsa X 3.1.1 - Atacar 3.2 - Si el jugador polsa tecla de moviment 3.2.1 - Moure personatge 3.3 - Actualitzar canvas 3.4 - Calcular col·lisions
Flux alternatiu:	**-**
Postcondició:	Tot el necessari per jugar s'ha carregat correctament.

Cas d'us Atacar	
Actor:	Registrat
Versió:	1.0
Descripció:	Llença un atac en la direcció on està mirant aquell personatge.
Precondició:	El personatge no està en moviment ni està atacant.
Flux principal:	1 - L'usuari polsa la tecla X 2 - El personatge queda bloquejat en estat de formulant conjur. 3 - Quan acaba l'animació de conjurar, el personatge dispara el conjur. 4 - Si impacta contra un personatge 4.1 - Resta vida al personatge enemic.
Flux alternatiu:	**-**
Postcondició:	L'acció d'atacar queda realitzada.

Cas d'us Moure personatge	
Actor:	Registrat
Versió:	1.0
Descripció:	Desplaça el personatge en la direcció depenent de la tecla polsada.
Precondició:	El personatge no està conjurant.
Flux principal:	1 - L'usuari polsa una tecla de moviment. 2 - El personatge fa l'animació de caminar. 3 - El personatge es mou.
Flux alternatiu:	**-**
Postcondició:	El personatge ha realitzat l'acció de moure's.

8.1.3. Servidor

En aquest apartat exposarem els casos d'ús que pot realitzar el servidor així com les seves fitxes de cas d'ús per veure en profunditat quines accions pot fer i com funcionen internament.

Figura 16. Casos d'ús servidor

Cas d'us Sincronitzar informació entre clients	
Actor:	Servidor
Versió:	1.0
Descripció:	Envia informació que rep d'un client a un altre.
Precondició:	El listener de sockets està en marxa.
Flux principal:	1 - Si el socket rep un missatge de xat 1.1 - Enviar missatge de xat 2- Si el socket rep un missatge del joc 2.1 - Enviar jugador remot
Flux alternatiu:	**-**
Postcondició:	El missatge s'ha enviat a tots els sockets de la sala menys a l'emissor del missatge.

Cas d'us Enviar jugador remot	
Actor:	Servidor
Versió:	1.0
Descripció:	Processa la informació del jugador i l'envia als altres sockets de la sala per que puguin sincronitzar el personatge.
Precondició:	El *listener* de sockets està en marxa.
Flux principal:	1 - Rep el *JSON* amb les dades del jugador. 2 - Processa les dades. 3 - Envia les dades a tots els *sockets* de la sala excepte l'emissor.
Flux alternatiu:	**-**
Postcondició:	Les dades han estat enviades.

Cas d'us Enviar missatge de xat	
Actor:	Servidor
Versió:	1.0
Descripció:	Rep un missatge, el processa i l'envia als altres sockets de la sala menys a l' emissor.
Precondició:	El listener de sockets està en marxa.
Flux principal:	1 - Rep un missatge de xat. 2 - Processa les dades 3 - Envia el missatge a tots els sockets menys a l'emissor.
Flux alternatiu:	**-**
Postcondició:	El missatge ha estat enviat.

8.1.3. Timer

En aquest apartat, exposarem els casos d'ús que pot realitzar el *timer* registrat, així com les seves fitxes de cas d'ús, per veure en profunditat quines accions pot fer i com funcionen internament.

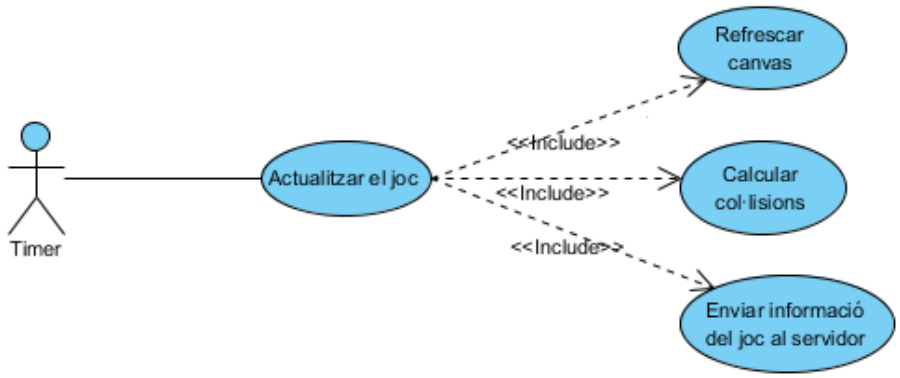

Figura 17. Casos d'ús timer

Cas d'us Actualitzar joc	
Actor:	Timer
Versió:	1.0
Descripció:	Refresca el canvas i calcula les col·lisions entre objectes.
Precondició:	La partida ha començat.
Flux principal:	1 - Mentre els dos personatges estan vius o queden dos personatges dintre de la partida 1.1 - Calcular col·lisions 1.2 - Envia la informació al servidor 1.3 - Refrescar canvas
Flux alternatiu:	-
Postcondició:	El joc queda actualitzat.

Cas d'us Calcular col·lisions	
Actor:	Timer
Versió:	1.0
Descripció:	Comprova si dos objectes han col·lidit i executa els processos pertinents.
Precondició:	La partida ha començat.
Flux principal:	1 - Comprova els grups de col·lisions 2 - Si un conjur impacta contra un personatge 2.1 - Li resta vida al personatge 3 - Si un personatge impacta contra un obstacle 3.1 - No pot avançar
Flux alternatiu:	-
Postcondició:	S'ha fet el tractament corresponent per cada grup de col·lisions.

Cas d'us Refrescar canvas	
Actor:	Timer
Versió:	1.0
Descripció:	Re-dibuixa tots els elements del *canvas*.
Precondició:	La partida ha començat.
Flux principal:	1 - Comprova les coordenades del mapa. 2 - Si el personatge s'ha mogut 2.1 - Dibuixa el mapa en la posició on està el centre de la càmara. 3 - Si el personatge ha disparat un conjur 3.1 - Dibuixa el projectil
Flux alternatiu:	-
Postcondició:	Tots els elements gràfics han estat actualitzats.

Cas d'us Enviar informació del joc al servidor	
Actor:	Timer
Versió:	1.0
Descripció:	Envia la informació al servidor.
Precondició:	La partida ha començat.
Flux principal:	1 - Processa les dades del jugador remot. 2 - Les converteix a JSON. 3 - Les envia a través del socket.
Flux alternatiu:	-
Postcondició:	L'informació és enviada.

8.2. Diagrama de seqüència de jugar

En aquest apartat exposarem el diagrama de seqüència de jugar, l'operació més complexa del nostre sistema. No mostrem altres diagrames de seqüència, perquè les altres operacions són molt simples i seria poc útil afegir-los.

Com el diagrama de seqüència és una mica dens. L'hem dividit en dues parts: la inicialització i jugar.

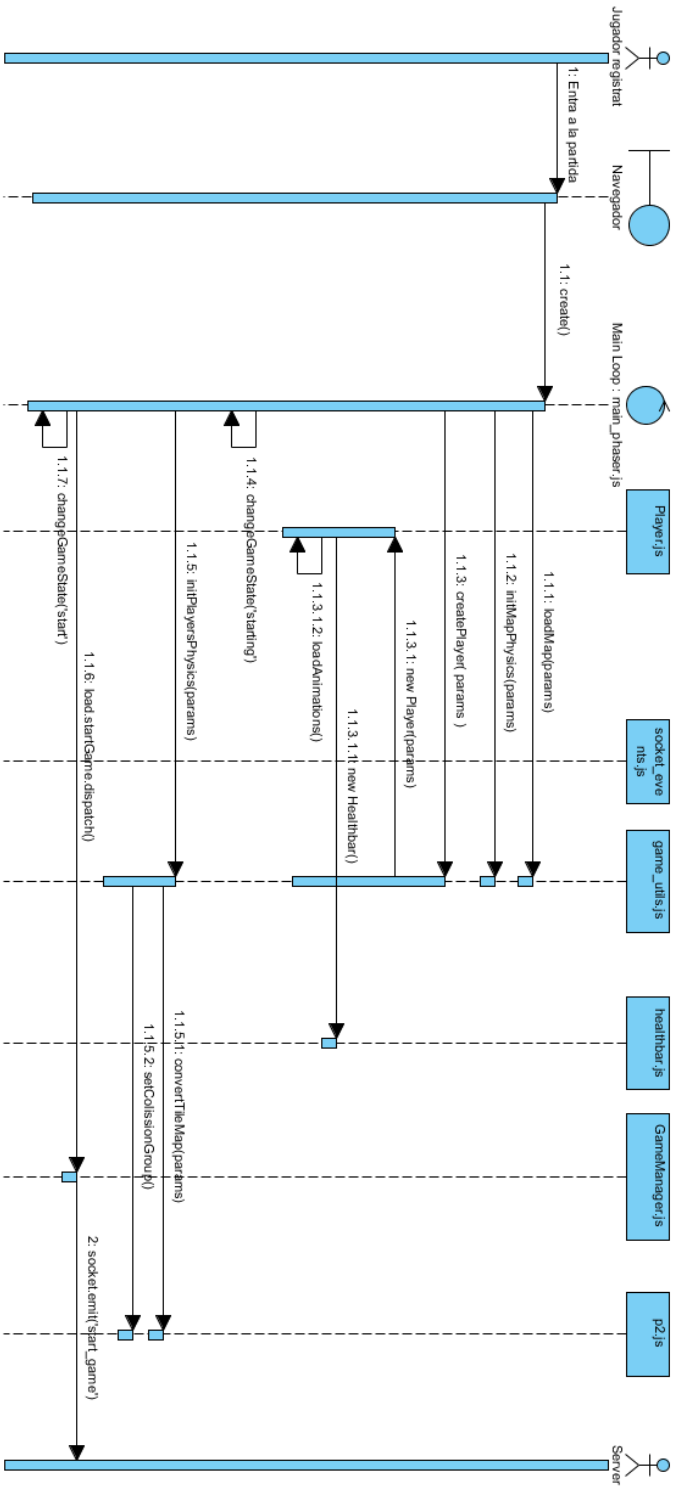

Figura 18. Diagrama de seqüència inicialització.

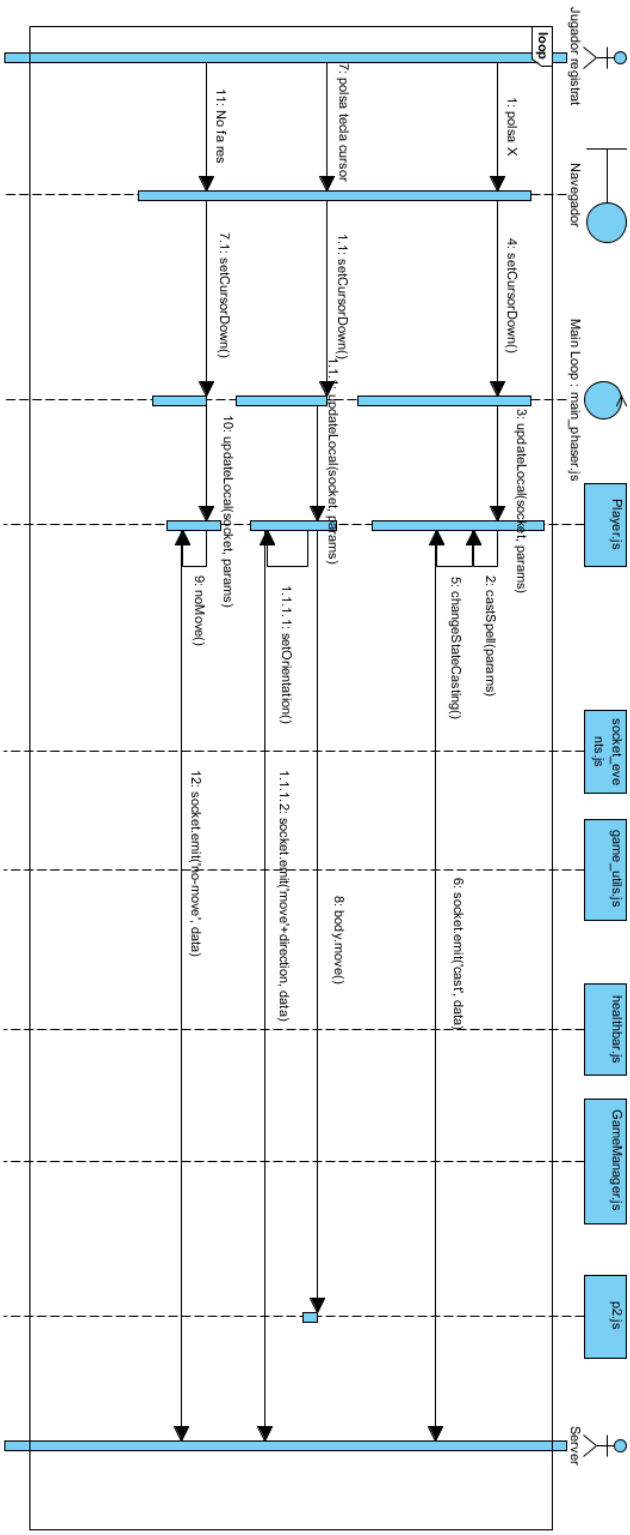

Figura 19. *Main Loop*

8.3. Diagrama de base de dades

Perquè la nostra aplicació funcioni només necessitem 3 col·leccions. Veurem que el següent diagrama no té relacions, així que aprofitaré per recordar que **mongodb** no és **relacional**.

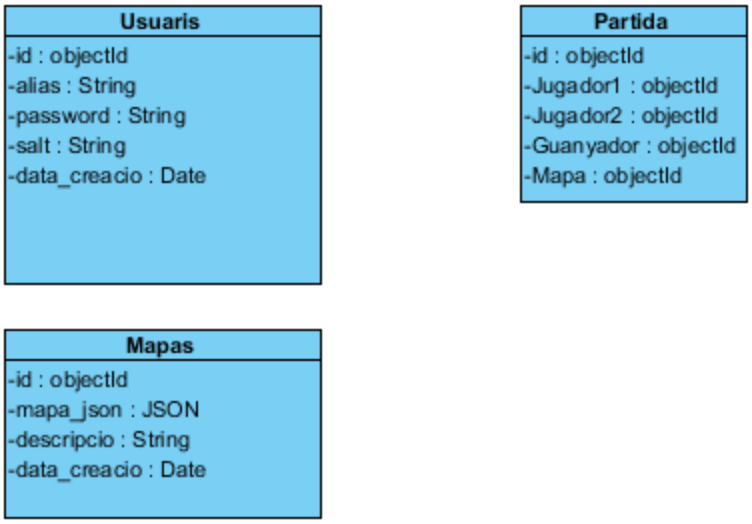

Figura 20. Diagrama de base de dades

En aquest diagrama veiem que tenim una col·lecció on guardem els usuaris registrats. Una col·lecció per guardar les partides, que farem servir per mostrar el rànquing. I una col·lecció de mapes on guardarem el *json* de l'atles que farem servir per carregar-lo.

8.4. Disseny de mòduls

En aquest apartat parlem de mòduls i no de classes. Això és així perquè en javascript no hi ha classes pròpiament. Hi ha una alternativa anomenada *pseudo-classes*. Per tant, hem optat per fer un disseny de mòduls per encapsular funcions per àmbits.

Per un costat explicarem els mòduls del client i per l'altre els del servidor.

8.4.1. Disseny mòduls client

A continuació exposarem un diagrama dels mòduls que tindrem al client i els explicarem breument. Podem observar que tots acaben en ".js" això ho he indicat així, perquè seran els mòduls que es crearan, no les classes.

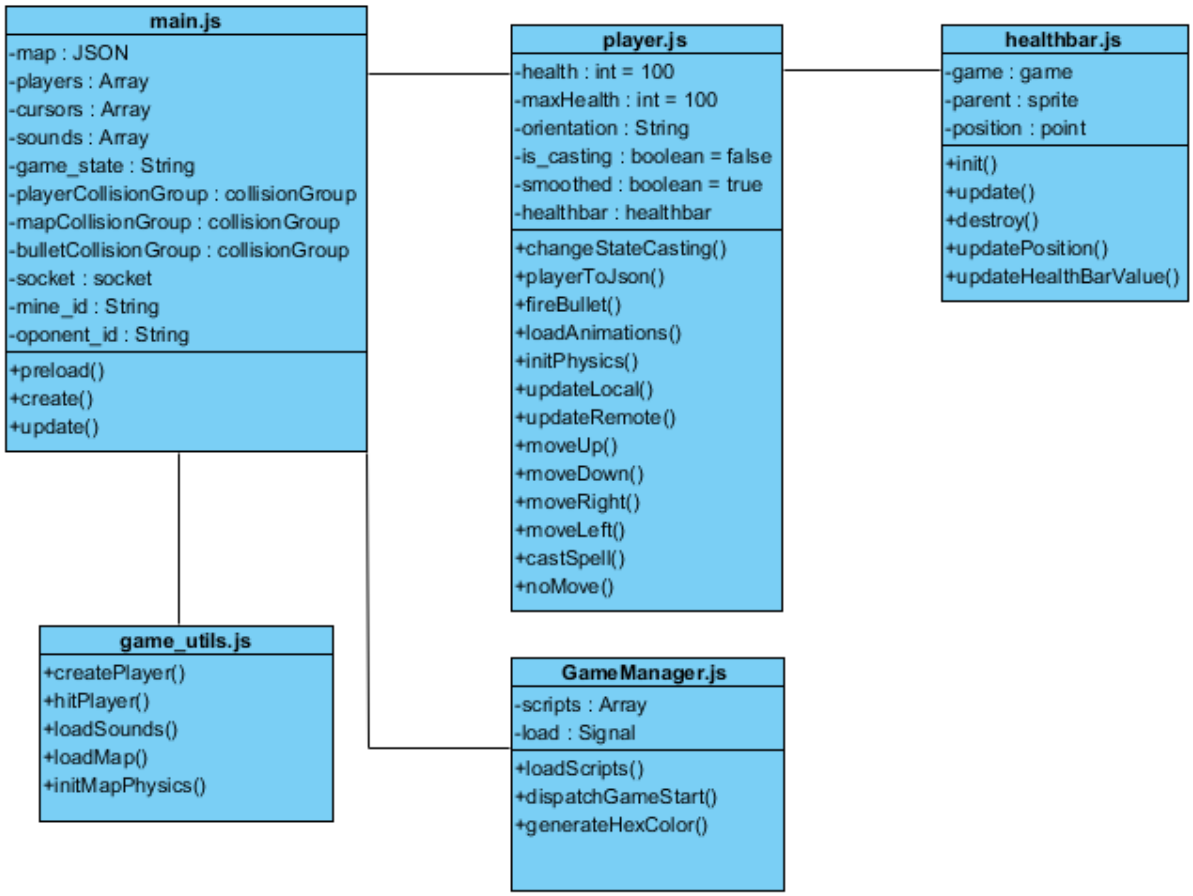

Figura 21. Mòduls part client

El disseny proposat es compon de 5 mòduls, tenint com a principal el main. El **main** és qui contindrà totes les variables globals necessàries per executar el joc. També tenim les tres funcions principals que s'executen en l'estat principal del joc. Com podem veure en el disseny tenim una col·lecció de *players*, així que encara que nosaltres hàgim limitat el joc a 2 jugadors, està preparat per permetre més.

El **game utils** el proposem com un mòdul a part per tal de separar funcions del main. Servirà per carregar i inicialitzar elements del joc.

El **game manager** serà una classe intermitja que la farem servir per carregar tots els mòduls que necessita el joc i disparar el event de començar el joc.

Per altra banda, tenim **player** i **healthbar**. Aquestes seran les dues **pseudo-classes** que tindrem com a tal. Aquestes les hem dissenyat així, perquè d'aquestes si que necessitarem varies instàncies. Tindran com a dependència que s'hagi arrancat bé el Phaserjs i s'hagi connectat correctament el socket, ja que aquest dos són necessaris per pintar el jugador local i per rebre les dades del jugador remot i actualizar-lo, així com enviar les dades del nostre jugador.

8.4.2. Disseny mòduls servidor

A continuació exposarem un diagrama dels mòduls que tindrem al servidor i els explicarem breument. Aquí també observarem que els fitxers acaben amb ".js" per el mateix motiu que en l'apartat anterior.

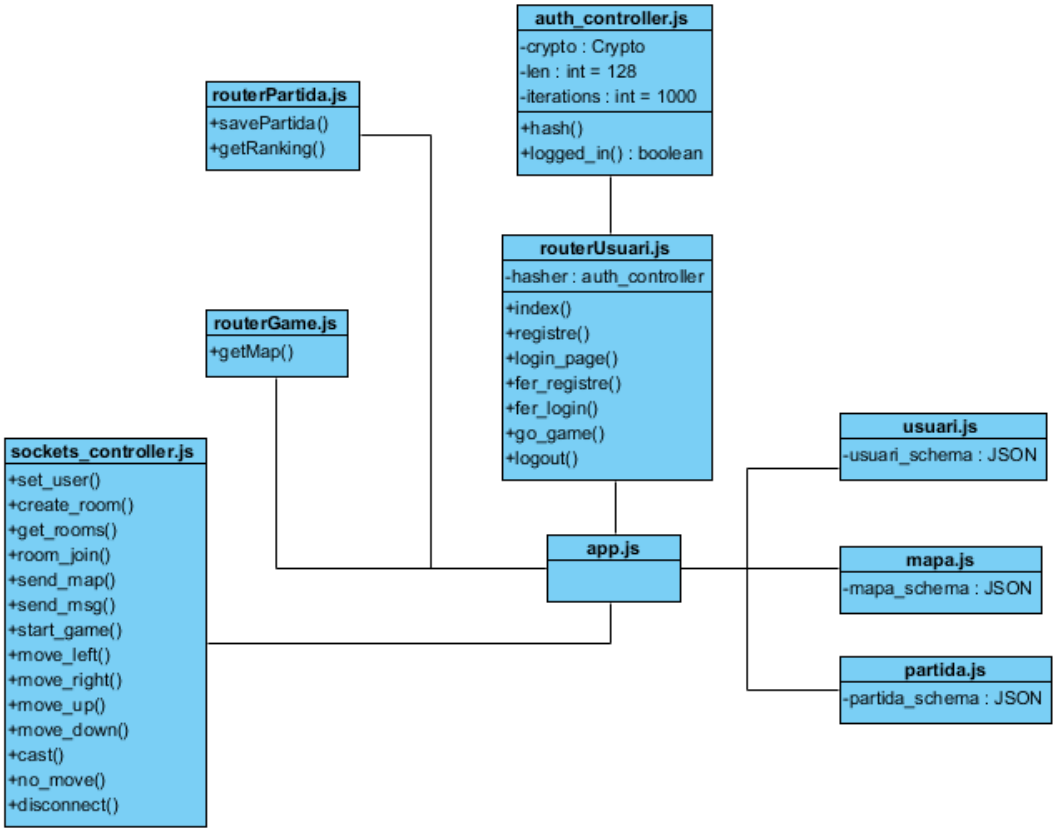

Figura 22. Mòduls part servidor

Com podem veure el disseny s'ha fet a partir de l'estructura de **nodejs,** que ens obliga a tenir un **app.js** que és des d'on arranca l'aplicació. A partir d'aquí hem fet una separació dels models de persistència. Aquests models són **usuari.js**, **mapa.js** i **partida.js**, podem veure que en els 3 casos l'únic que tindrem serà l'esquema de dades.

Després hem separat per sectors les funcions que farem servir per a l'encaminador. Com podem veure a la figura en tenim tres. Com a especial tenim el **routerUsuari.js** que té una instància del **auth_controller.js** que ens servirà per encriptar passwords i com a auxiliar per saber si l'usuari està autentificat correctament.

Per últim, tindríem un mòdul que encapsularà totes les funcions que es faran servir en el servidor de sockets. Aquestes serien les mínimes indispensables per a què el joc es pugui comunicar.

8.5. Especificació dels missatges de sockets

En aquest apartat mostraré l'especificació dels missatges del socket, quin event estan escoltant i quin event enviaran de resposta. Així com les dades que rebran i les que enviaran. Hem volgut fer aquesta especificació per la semblança que té amb les Api REST [37].

Per la part del servidor tindrem:

Event que escolta	Dades que rep	Event que emet	Dades que envia
connection	L'objecte socket de la llibreria.	connection_confirmation	Missatge de connexió correcte.
set_user	El usuari *logat.*	-	-
create_room	El nom de la sala.	confirm_room_creation	Missatge de sala creada correctament.
get_rooms	-	room_list	Llistat de les sales que tenim actives al servidor.
room_join	Nom de la sala a la qual ens volem unir.	sincronize_map	-
send_msg	El missatge que envia l'usuari.	recive_msg	L' àlies de l'usuari emissor i el missatge.
sincronize_map	El id del mapa.	confirm_room_join	Missatge de ok o ko i si tot ha anat

			bé, enviem el id del mapa.
start_game	-	get_mate_player	El id de socket i l'usuari que està guardat.
move_up	El personatge en format json.	move_up	El personatge en format json.
move_down	El personatge en format json.	move_down	El personatge en format json.
move_right	El personatge en format json.	move_right	El personatge en format json.
move_left	El personatge en format json.	move_left	El personatge en format json.
cast	El personatge en format json.	cast	El personatge en format json.
no_move	El personatge en format json.	no_move	El personatge en format json.
disconnect	-	player_disconnected	-

Per la part del client tindrem:

Event que escolta	Dades que rep	Event que emet	Dades que envia
connection_confirmation	Missatge de connexió correcte.	get_rooms	-
confirm_room_creation	Missatge de sala creada correctament.	-	-
room_list	Llistat de les sales que tenim actives al servidor.	-	-

sincronize_map	-	sincronize_map	El id del mapa.
confirm_room_join	Missatge de ok o ko i si tot ha anat bé, enviem el id del mapa.	-	-
get_mate_player	L' estat inicial de personatge rival.	start_game	
move_left	El personatge en format json.	-	-
move_up	El personatge en format json.	-	-
move_right	El personatge en format json.	-	-
move_down	El personatge en format json.	-	-
cast	El personatge en format json.	-	-
no_move	El personatge en format json.	-	-
player_disconnected	-	-	-
recive_msg	El missatge del xat.	-	-

9. Implementació i proves

En aquest apartat explicarem els detalles de la implementació de la nostra aplicació. Ho separarem en 2 blocs: el client i el servidor.

9.1. Client

La part del client s'ha implementat amb javascript, html i css. Com podem veure a la següent figura, hem definit 4 pàgines principals:

1. El login: on l'usuari accedeix a l'aplicació, també és la pàgina principal. Si accedeix correctament passa a la pàgina del joc.
2. El registre: on l'usuari fa el registre i si es registra correctament, accedeix automàticament a la pàgina del joc.
3. El rànquing: on l'usuari visualitza el rànquing de victòries, pot accedit a la pàgina del joc si ja està logat amb anterioritat.
4. La pàgina del joc: on l'usuari pot crear o unir-se a una partida i posteriorment, per *AJAX*, es carreguen les dades necessàries per iniciar el joc. És la mateixa pàgina on jugarà i xatejarà també.

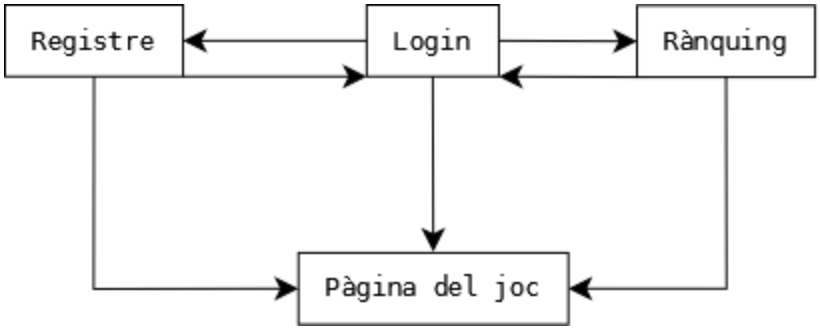

Figura 23. Esquema navegació web

Per cada vista tenim una plantilla, i per cada plantilla tenim una ruta. Aquestes rutes les veurem en el punt 9.2.2 on explicarem totes les rutes del servidor. Tant les que són per GET i per POST.

9.1.1. Estructura de directoris

L'estructura de directoris del client és la següent:

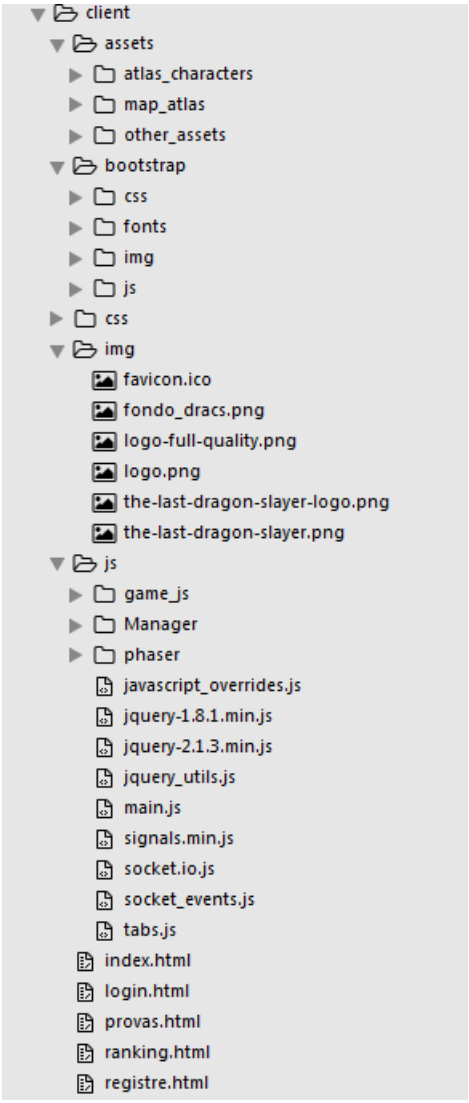

Figura 24. Estructura de directoris part client

En aquesta estructura podem veure com tenim 4 grans blocs:

1. assets: Són els **arxius multimèdia del joc**, com per exemple els sons, les imatges dels atles, etc.
2. css: Aquí tenim els arxius que serveixen per **donar estils** al nostre web.
3. img: En aquesta carpeta tenim les **imatges** del web pròpiament, les que no tenen res a veure amb el joc.

4. js: Aquí tenim agrupats tots els **javascripts** que utilitzarem al web i al joc. Hem fet la següent subdivisió:
    a. game_js: Aquí tenim l'arxiu del **main loop** i les **classes del joc**.
    b. manager: Hem separat el manager en una altra carpeta, perquè realment fa de pont entre el web i el joc.

Com podem veure també tenim una plantilla per cada vista que hem exposat en l'apartat anterior.

9.1.2. Mòduls del joc

En aquest apartat explicarem en detall la implementació dels mòduls del joc. El mòdul principal és el **main.js.** En aquest mòdul és on tenim tota la lògica d'inicialització del joc.

```
preload: function()
{
    //Carreguem els atlas i el demanem el json del mapa
    game.load.tilemap('map', '/get_mapa/'+socket.id_game_map, null, Phaser.Tilemap.TILED_JSON);
    game.load.image('base_out_atlas', 'assets/map_atlas/base_out_atlas.png', 32, 32);
    game.load.image('terrain_atlas', 'assets/map_atlas/terrain_atlas.png', 32, 32);
    game.load.image('farming_fishing', 'assets/map_atlas/farming_fishing.png', 32, 32);
    //Carreguem els atlas del personatges
    game.load.atlasJSONHash('girl', 'assets/atlas_characters/girl/sprites.png', 'assets/atlas_characters/girl/sprites.json
    game.load.image('healthbar', 'assets/other_assets/healthbar.png');
    game.load.image('healthbar_frame', 'assets/other_assets/healthbar_frame.png');
    //Carreguem la imatge dels bullets
    game.load.image('bullet', 'assets/other_assets/bullet.png');
    //Carreguem els audios
    game.load.audio('battle_main_theme', 'assets/other_assets/battle_theme.mp3');
    game.load.audio('explosion', 'assets/other_assets/explosion.ogg');
    game.load.audio('casting', 'assets/other_assets/casting.ogg');
    game.load.audio('shot', 'assets/other_assets/shot.ogg');
},
create: function()
{
    game.stage.backgroundColor = '#308fe3';
    game.stage.disableVisibilityChange = true;

    //inicialitzem les tecles
    cursors = game.input.keyboard.createCursorKeys();
    extra_cursors['X'] = game.input.keyboard.addKey(Phaser.Keyboard.X);

    map = loadMap( game, layers );
    game.world.setBounds(0, 0, map.widthInPixels, map.heightInPixels);
    //Arranquem les fisiques i creem el personatge

    worldCollisionGroup = game.physics.p2.createCollisionGroup();
    playerCollisionGroup = game.physics.p2.createCollisionGroup();
    bulletCollisionGroup = game.physics.p2.createCollisionGroup();


    initMapPhysics( game, layers, map, worldCollisionGroup, playerCollisionGroup );
    game.physics.p2.updateBoundsCollisionGroup();

    createPlayer( mine_id, game, 600, 600, 'right', true, players, playerCollisionGroup );

    bullets = game.add.group();
    bullets.enableBody = true;
    bullets.physicsBodyType = Phaser.Physics.P2JS;

    game_start = false;

    loadSounds(sounds);
    sounds['battle_theme'].play('',0,1,true,true);
    gameText = game.add.text(10, 10, "Esperant oponent" ,  { font: "32px Arial", fill: GameManager.generateHexColor() });
    gameText.fixedToCamera = true;
    //textGroup.add(game.make.text(100, 64, 'here is a colored line of text',  { font: "32px Arial", fill: GameManager.gen
    GameManager.load.gameStart.dispatch();
},
update: function()
```

Figura 25. Estat principal del joc

A la figura anterior podem veure que en el main.js tenim els 3 estats principals del joc. El **preload** que com veiem només serveix per carregar els assets. El **create** que ens inicialitza el mapa, personatge i col·lisions. I per acabar el **update** que és el que s'executarà a cada *frame*.

No obstant és en els mòduls que veurem a continuació on estan totes les funcions necessàries, perquè el joc funcioni correctament.

9.1.2.1. Player.js i healthbar.js

El **player.js** i el **healthbar.js** són les úniques **pseudo-classes** que tenim. Ho hem fet així, perquè són els elements que s'han de dibuixar més complexes.

La **healthbar** realment és un atribut de **player,** ja que per ella mateixa no té cap sentit. Com veurem a la imatge següent, en la creació de la instància de *player* es crea la *healthbar.* També veurem que a part d'estendre de la classe Phaser.Sprite, crida al seu constructor.

```
//Creacio de la pseudo-class per el player
var Player = function(  game, x, y, orientation, sprite )
{
    //Inicialitzem el player com a objecte del phaser
    Phaser.Sprite.call( this, game, x, y, sprite );

    //assignacio de atributs externs al phaser
    this.health = 100;
    this.maxHealth = 100;
    this.orientation = orientation;
    this.is_casting = false;
    this.smoothed = true;
    //carreguem les animacions
    this.loadAnimations();

    //afegim la barra de vida
    this.healthbar = new HealthBar(game, this);
    this.healthbar.init(this.health);
    this.healthbar.visible = true;

    game.add.existing(this);
}

Player.prototype = Object.create(Phaser.Sprite.prototype);
Player.prototype.constructor = Player;
```

Figura 26. Inicialització de player

En la imatge veiem clarament com encadenem el ***prototype*** de *player* amb el de *Phaser.sprite*. A més a més, amb el Phaser.Sprite.call [38] fem una crida al constructor pare.

La healthbar per la seva banda encapsula totes les funcions necessàries per poder-se dibuixar. En la següent figura podrem apreciar-ho millor:

```
HealthBar.prototype.init = function (maxHealth) {
    this.healthBarFrame = this.game.make.sprite(0, 0, 'healthbar_frame');
    this._healthBar = this.game.make.sprite(0, 0, 'healthbar');
    this.healthBarFrame.visible = false;
    this._healthBar.visible = false;
    this.healthBarFrame.texture.baseTexture.scaleMode = PIXI.scaleModes.NEAREST;
    this._healthBar.texture.baseTexture.scaleMode = PIXI.scaleModes.NEAREST;
    this.healthBarFrame.renderPriority = 2;
    this._healthBar.renderPriority = 1;
    this.game.world.add(this.healthBarFrame);
    this.game.world.add(this._healthBar);
    this.visible = false;
    this.fullWidth = this._healthBar.width;
    this.maxHealth = maxHealth || 1;
    this.updateHealthBarValue();
    this.updatePosition();
};
HealthBar.prototype.update = function () {
    if (this._visible) {
        this.updatePosition();
        this.updateHealthBarValue();
    }
};
HealthBar.prototype.destroy = function () {
    this.visible = false;
    this.healthBarFrame.destroy();
    this._healthBar.destroy();
};
HealthBar.prototype.updatePosition = function () {
    var parent = this.parent;
    var x = parent.position.x - (28 * parent.anchor.x);
    var y = parent.position.y - (40 * parent.anchor.y) - this.healthBarFrame.height - 5;
    this.position.set(x, y);
    this.healthBarFrame.position.set(x, y);
    this._healthBar.position.set(x, y);
    //this._healthBar.bringToTop();
    //this.healthBarFrame.bringToTop();
};
HealthBar.prototype.updateHealthBarValue = function () {
```

Figura 27. Funcions principals de healthbar

9.1.2.2. Sockets

El funcionament de sockets és una de les parts més importants de la nostra aplicació. Quan l'usuari fa el login automàticament el socket fa la connexió.

El socket és un objecte que està subscrit a certs events i quan els rep, executa una funció de callback i opcionalment emet un event cap al servidor.

Com podem veure a la següent figura el socket està esperant un event i pot emetre o no un altre event cap al servidor:

```
socket.on('connection_confirmation', function(data)
{
    var usuari = JSON.parse($('#alias_usuari').val());
    socket.emit('set_user', { user: usuari });
    socket.emit('get_rooms', {});
    interval = setInterval(function(){socket.emit('get_rooms', {});},10000);
});

socket.on('confirm_room_creation',function(data)
{
    $('#main_actions').hide();
    $('#index-wrapper').css('visibility','visible');
    GameManager.loadScripts(socket);
});
```

Figura 28. Exemple de sockets.

9.2. Servidor

En la nostra implementació el servidor es l'encarregat de tres operacions al iniciar-se. Aquestes operacions són les següents:

1. Connectar amb la base de dades
2. Crear el servidor HTTP.
3. Crear el servidor de sockets

En la següent imatge exposarem els passos descrits:

```
//conectem la base de dades, en aquest cas utilitzem una db no-sql: mongodb
var Schema = mongoose.Schema;
mongoose.connect('mongodb://localhost:27017/tlds', function(err){if(err) {throw err;}else{console.log('Running mongoose version %s'
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', function callback () {
    console.log('DATABASE '+db.name+' CONNECTION SUCCESS');
});

//inicialitzem els models, els objectes que farem servir per obtenir dades de db.
require('./models/usuari')( Schema, mongoose );
require('./models/mapa')( Schema, mongoose );
require('./models/partida')( Schema, mongoose );
```

Figura 29. Inici de la base de dades i inicialització dels esquemes de dades.

```
//ROUTER
//Definim les rutes que tindra la nostre aplicacio.
var routerUsuari = require('./routes/routerUsuari')(mongoose);
app.get('/', routerUsuari.index);
app.get('/registrar', routerUsuari.registre );
app.get('/logout', routerUsuari.logout);
app.post('/fer-registre', routerUsuari.fer_registre);
app.post('/fer-login', routerUsuari.fer_login);
app.get('/go-game', routerUsuari.go_game);
var routerGame = require('./routes/routerGame')(mongoose);
app.get('/get_mapa/:id_mapa', routerGame.getMap);

var routerPartida = require('./routes/routerPartida')(mongoose);
app.get('/ranking', routerPartida.getRanking);
app.post('/save-match', routerPartida.savePartida)
//FI ROUTERS

//Init server on port 8080
server.listen(8080);
```

Figura 30. Inicialització de les rutes i del servidor HTTP.

```
io.sockets.on('connection', function(socket)
{
  //inicialitzem el objecte que conte totes les funcions que tracta les peticions dels sockets.
  var socket_functions = require('./controller/socket_controller')(socket, io);
  //si conecta correctament li enviem un missatge.
  socket.emit('connection_confirmation', { msg: 'Connection success!' });
  //quan rebem aquest event setejem el alias al socket.
  socket.on('set_user', socket_functions.set_user);
  //quan rebem aquest event creem la room.
  socket.on('create_room', socket_functions.create_room);
  //quan rebem aquest event enviem les rooms amb menys de 2 sockets conectats.
  socket.on('get_rooms', socket_functions.get_rooms);
  //quan rebem aquest event, afegim el socket actual a la room
  socket.on('room_join', socket_functions.room_join);
  //funcio que envia el missatge a els demes sockets conectats a la sala.
  socket.on('send_msg', socket_functions.send_msg);
  //funcio que envia el mapa al player 2
  socket.on('sincronize_map', socket_functions.send_map);
  //quan rebem aquest event, enviem les dades necesaries per començar el joc.
  socket.on('start_game', socket_functions.start_game);
  //quan rebem aquest event, enviem el personatge en json.
  socket.on('move_up', socket_functions.move_up);
  //quan rebem aquest event, enviem el personatge en json.
  socket.on('move_down', socket_functions.move_down);
  //quan rebem aquest event, enviem el personatge en json.
  socket.on('move_right', socket_functions.move_right);
  //quan rebem aquest event, enviem el personatge en json.
  socket.on('move_left', socket_functions.move_left);
  //quan rebem aquest event, enviem el personatge en json.
  socket.on('cast', socket_functions.cast);
  //quan rebem aquest event, enviem el personatge en json.
  socket.on('no_move', socket_functions.no_move);
  //quan rebem aquest event executem el necesari per acabar la partida o desconectar el socket normalment.
  socket.on('disconnect', socket_functions.disconnect);
});
```

Figura 31. Inicialització del servidor de sockets

9.2.1. Estructura de directoris

L'estructura de directoris del servidor és la següent:

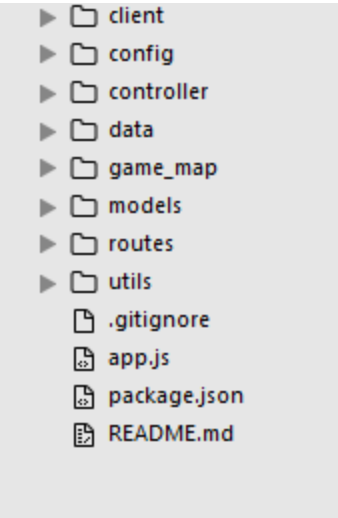

Figura 32. Estructura de directoris del servidor

Aquí podem veure que hem separat els fitxers per:

1. client: és el que hem vist al punt 9.1.1.
2. config: configuració addicional del servidor.
3. data: és la carpeta on es guarden les col·leccions de mongodb.
4. game_map: tenim els fitxers d'importació de mapes a base de dades.
5. models: són els models de persistència a base de dades.
6. routes: són les funcions que farem servir en l'encaminador de peticions HTTP.
7. utils: funcions genèriques d'utilitats.

En els següents punts veurem millor explicat el contingut d'aquests directoris.

9.2.2. Models

Encara que diem que **mongodb** no té esquema de dades, amb la llibreria **moongose** hem pogut simular que en té.

Hem implementat 3 models descrits en l'apartat de disseny. El de mapa, el d'usuari i el de partida. A continuació veurem un exemple de com són aquests:

```
module.exports = function( Schema, odm )
{
    console.log('Creant el esquema de mapa');
    var mapa_schema = Schema({
        mapa_json: Schema.Types.Mixed,
        descripcio: String,
        data_creacio: { type: Date, default: Date.now }
    });

    odm.model('Mapa', mapa_schema);
};
```

Figura 33. Model de mapa

En el següent model podem apreciar com creem l'esquema, i com l'afegim al **odm** (*object document mapper*). Gràcies a aquests models el primer cop que inserim un document a la col·lecció si no existeix, ens la crea també.

Els models també ens serveixen per fer consultes a base de dades, ja que cada model disposa de les funcions natives de mongodb per fer cerques a la col·lecció que apunten.

Per cada model hem definit un encaminador sobre el qual es farà una petició HTTP i aquestes funcions realitzaran les funcions **CRUD**[19].

9.2.3. Mòduls de l'encaminador

Com hem vist en apartats anteriors tenim 3 mòduls d'encaminadors. Cada mòdul l'hem separat segons sobre quin model treballava. Com hem vist en l'apartat anterior,

[19] CRUD: create, read, update, delete

quan inicia les rutes aquestes tenen una funció de *callback*. Aquesta funció està encapsulada dintre d'un mòdul d'aquests que hem creat.

Cada funció rep com a paràmetres la *request* i la *response*. En la request és on tenim totes les dades de **sessió** i de **cookies**. La *response* és l'objecte que farem servir per retornar-la cap al client.

En la següent imatge podrem apreciar el funcionament:

```
getMap: function( req, res )
{
    var Mapa = mongoose.model('Mapa');
    var ObjectId = mongoose.Types.ObjectId;

    var idMapa = new ObjectId(req.params.id_mapa);

    Mapa.findById( idMapa, function( error, result )
    {
        if(error)
            throw error;

        res.end(JSON.stringify(result.mapa_json));

    });
}
```

Figura 34. Exemple de funció de ruta

Podem veure com cridem la funció, agafem els paràmetres de la request que en aquest cas ens arriben per POST. Fem una consulta a base de dades. I amb el *response.end* enviem el JSON de resposta cap al client.

En la següent figura veurem el que rebem a la part de client quan fem aquest procés:

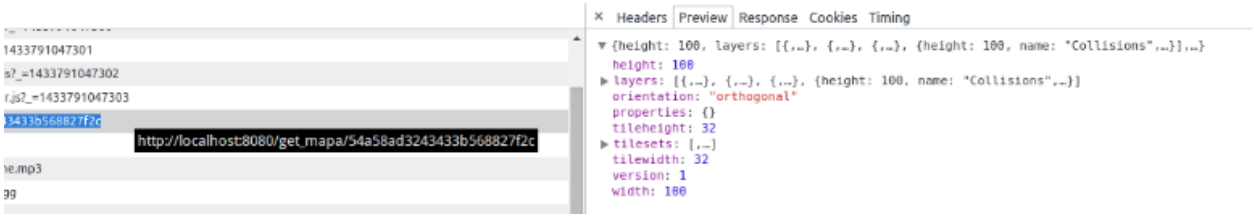

Figura 35. Mostra del retorn del get_mapa

9.2.4. Controladors

Hem creat dos controladors. Els controladors en el nostre cas són funcions de control. En el cas del auth_controller controlem la seguretat de la nostra aplicació amb dues operacions, una per encriptar la contrasenya de l'usuari i l'altre per comprovar en la sessió que l'usuari hagi accedit correctament.

```
exports.hash = function (pwd, salt, fn) {
    if (3 == arguments.length) {
        crypto.pbkdf2(pwd, salt, iterations, len, function(err, hash){
            fn(err, (new Buffer(hash, 'binary')).toString('base64'));
        });
    } else {
        fn = salt;
        crypto.randomBytes(len, function(err, salt){
            if (err) return fn(err);
            salt = salt.toString('base64');
            crypto.pbkdf2(pwd, salt, iterations, len, function(err, hash){
                if (err) return fn(err);
                fn(null, salt, (new Buffer(hash, 'binary')).toString('base64'));
            });
        });
    }
};

exports.logged_in = function( req )
{
    if( !javascript_utils.empty(req.session.user) || !javascript_utils.empty(req.cookies.user) )
    {
        //si queda a les cookies i a la sessio no. Guardem el alias en sessio.
        if( javascript_utils.empty(req.session.user) )
        {
            req.session.user = req.cookies.user;
        }
        return true;
    }
    else
    {
        return false;
    }


};
```

Figura 36. Controlador de autentificació

L'altre controlador que tenim és el de sockets. Aquest controlador l'hem anomenat com a tal, perquè realment controla el flux de dades dels sockets i envia les dades pertinents al client. En la següent figura veurem un exemple:

```
get_rooms : function(data)
{

    var room_list = [];

    for( var room in io.sockets.manager.rooms )
    {
        //Si hi ha menys de 2 players no mostrem la sala. Esta plena.
        if( io.sockets.manager.rooms[room].length < 2 )
            room_list.push(room);
    }
    socket.emit('room_list', {room_list: room_list});
},
```

Figura 37. Exemple de funció del socket controller

Com podem veure aquesta funció agafa unes dades les processa i les envia en un event cap al client.

10. Implantació i resultats

En aquest apartat explicaré la senzillesa de posar en marxa el servidor de **Nodejs** i mostraré els resultats obtinguts després de la fase de desenvolupament.

10.1. Posar en marxa el servidor

Per posar en marxa el servidor és molt senzill. Només ens cal tenir instal·lat el **Nodejs.** Això és així perquè Nodejs ja té incorporat un gestor de paquets el npm [39]. Així que nosaltres només tindríem que executar la comanda “npm install” i ell ja instal·laria les dependències necessàries per executar la nostra aplicació.

Les dependències les llegeix d’un fitxer anomenat “package.json” que té un format concret. En ell posem els paquets que necessitem i fins i tot podem especificar la versió que volem. En la següent figura s’il·lustra millor l’estructura d’aquest fitxer.

```
{
  "description": "joc pvp",
  "version": "0.0.0-6",
  "private": true,
  "dependencies": {
    "express": "3.x",
    "express-ejs-layouts": "*",
    "ejs": "*",
    "mongodb": "1.3.0",
    "mongoose": "*",
    "socket.io": "0.9.x",
    "static": "*",
    "signals" : "*"
  },
  "devDependencies": {
    "api-easy": "0.3.2",
    "vows": "0.6.x"
  },
  "scripts": {
    "test": "vows --spec",
    "start": "node app.js"
  },
  "homepage": "",
  "author": "Archel",
  "name": "tlds",
  "subdomain": "tlds",
  "engines": {
    "node": "0.8.x"
  }
}
```

Figura 38. package.json del nostre projecte

Un cop instal·lades les dependències executaríem la comanda “npm start” i ja tindríem el servidor funcionant en un entorn de proves.

```
archel@Midgar:~/proc/tlds$ npm start

> tlds@0.0.0-6 start /home/archel/proc/tlds
> node app.js

   info  - socket.io started
Creant el esquema de usuari
Creant el esquema de mapa
Creant el esquema de partida
Inicialitzan les funcions del router de usuari
ok, server is running
Running mongoose version 4.0.1
DATABASE tlds CONNECTION SUCCESS
```

Figura 39. *Output* de la comanda “npm start”

10.2. Resultats obtinguts

A continuació mostrarem amb imatges i una breu explicació els següents resultats:

En la següent imatge veiem la home que ens permet accedir a l’aplicació, anar al registre o veure el rànquing.

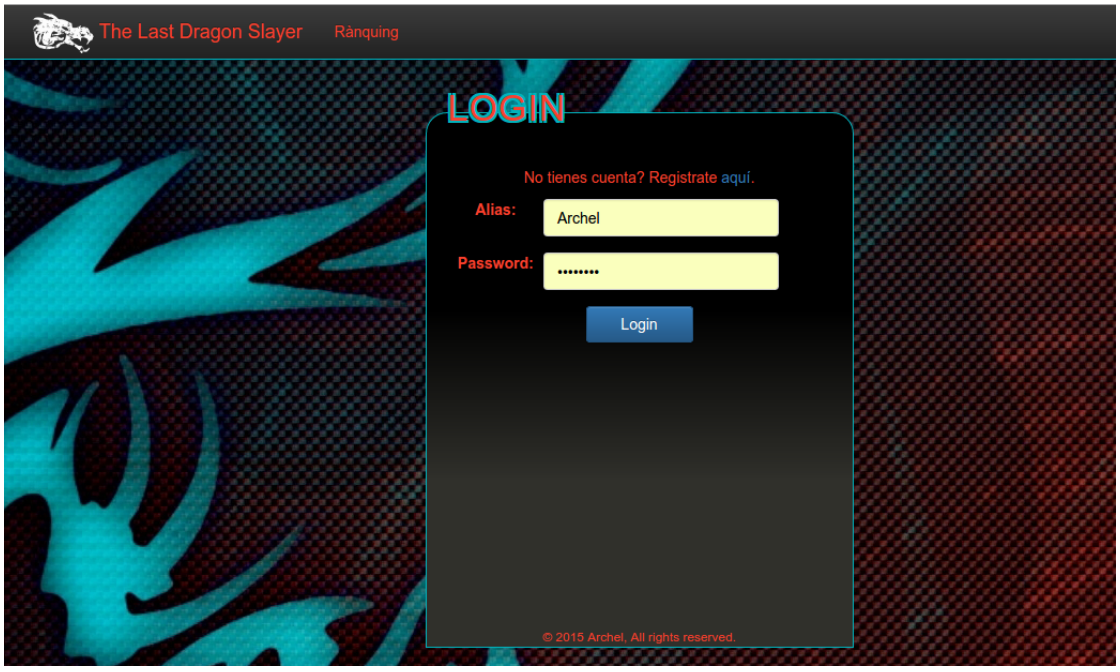

Figura 40. Login

Si tenim error d'usuari o contrasenya es mostra un error per pantalla amb una alerta:

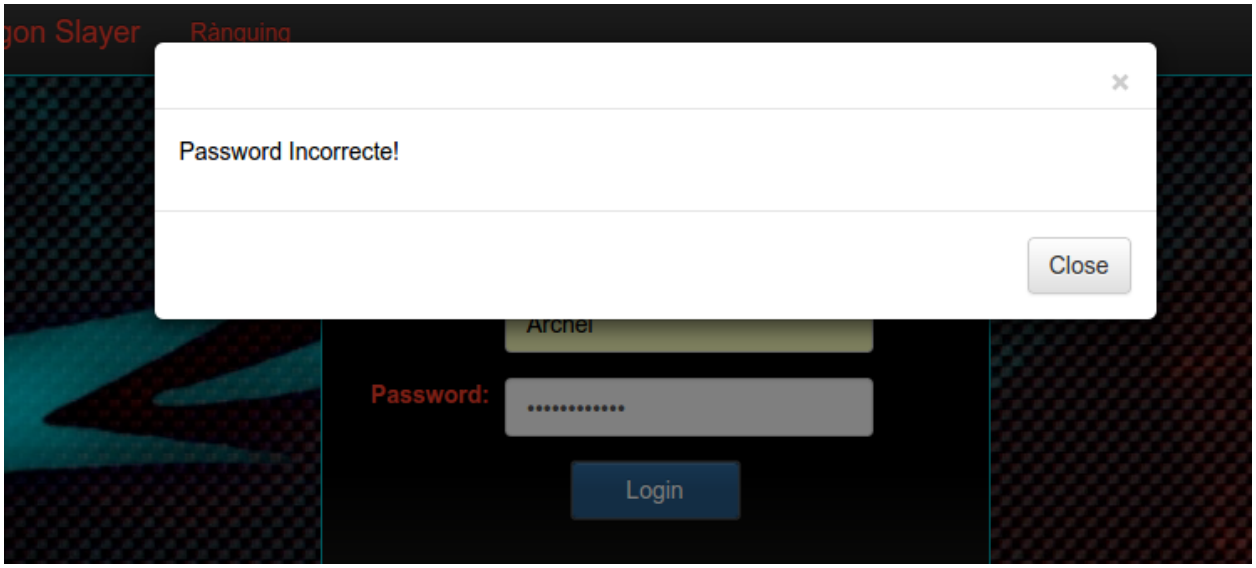

Figura 41. Error en la contrasenya

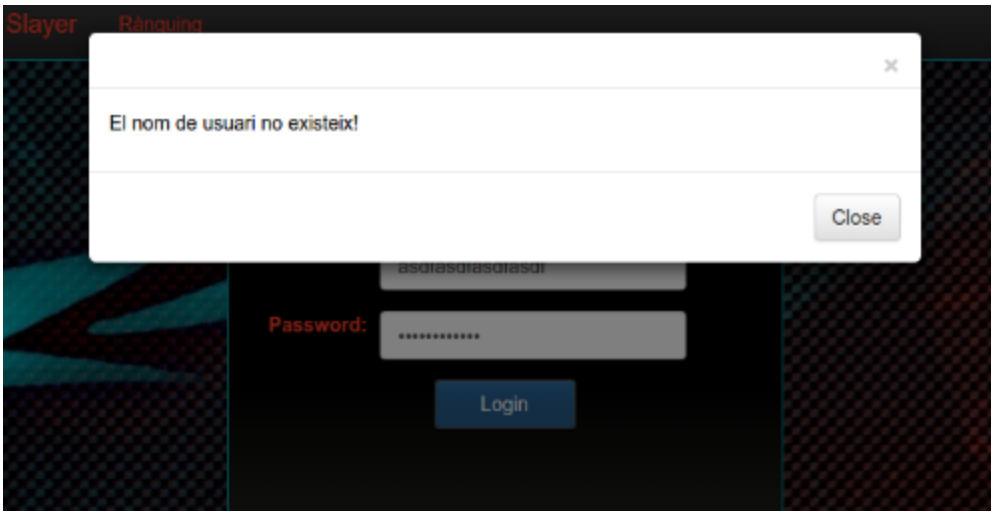

Figura 42. Error en el nom d'usuari

El formulari de registre és el mateix que el del login però amb els textos canviats per tant només afegiré la foto de quan intentem registrar-nos i el nom d'usuari ja existeix:

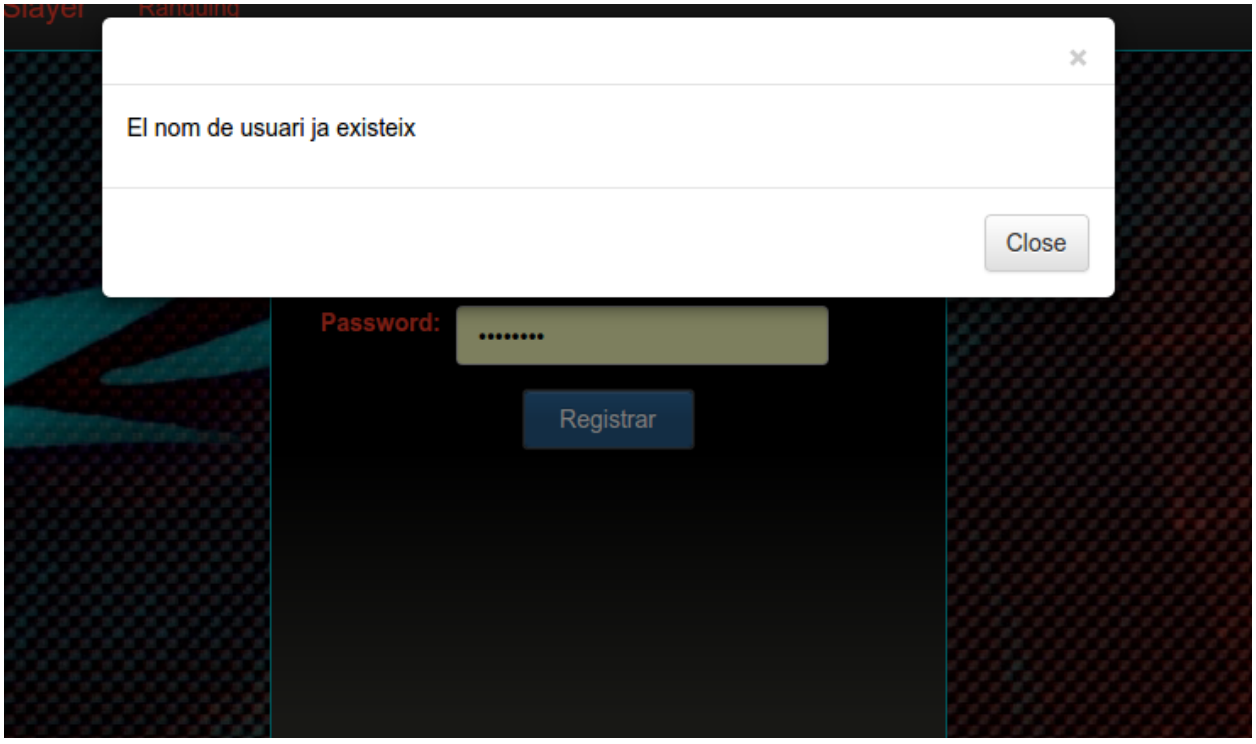

Figura 43. Error en el registre

Per acabar amb les pàgines de la part pública mostrarem el rànquing:

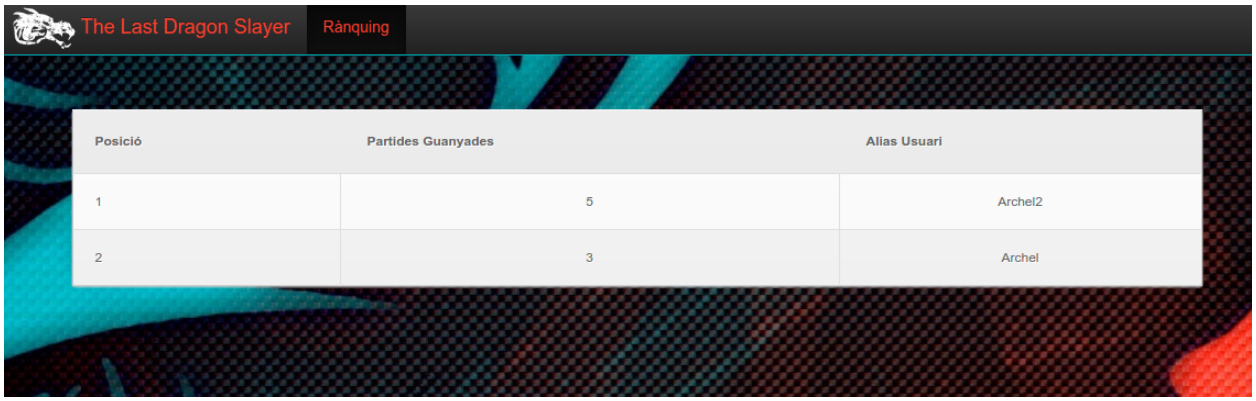

Figura 44. Rànquing

A continuació veurem el procés de crear una partida:

1. Posem el nom a la partida i seleccionem un mapa

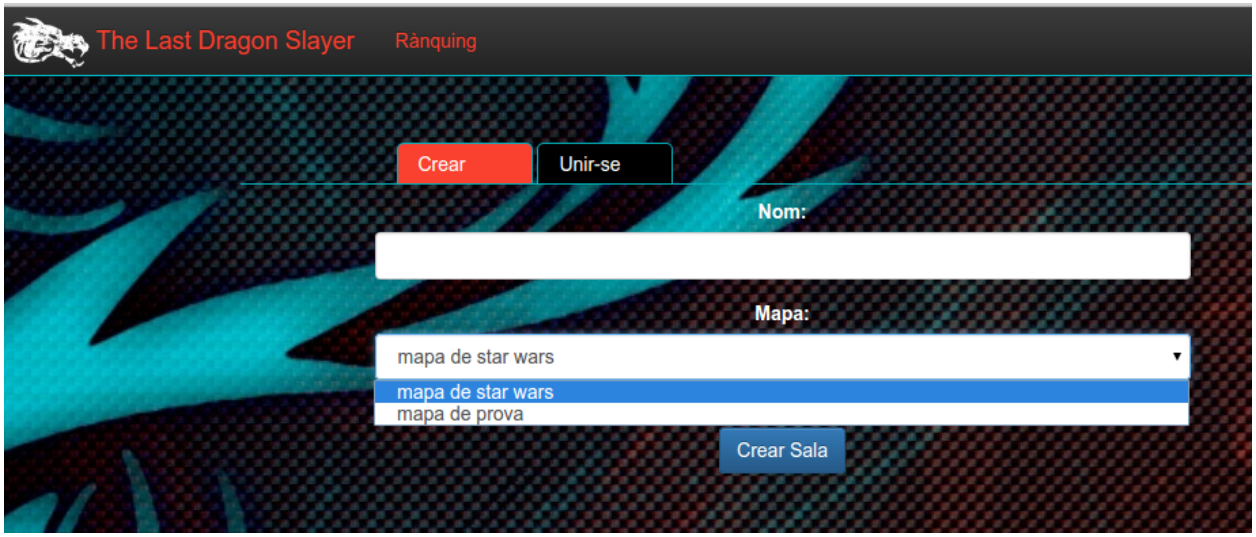

Figura 45. Pantalla de creació de partida

2. Es carrega el joc i es queda en estat d'espera.

Figura 46. Pantalla de joc, esperant oponent

Si fem el mateix procés però en comptes de crear ens unim:

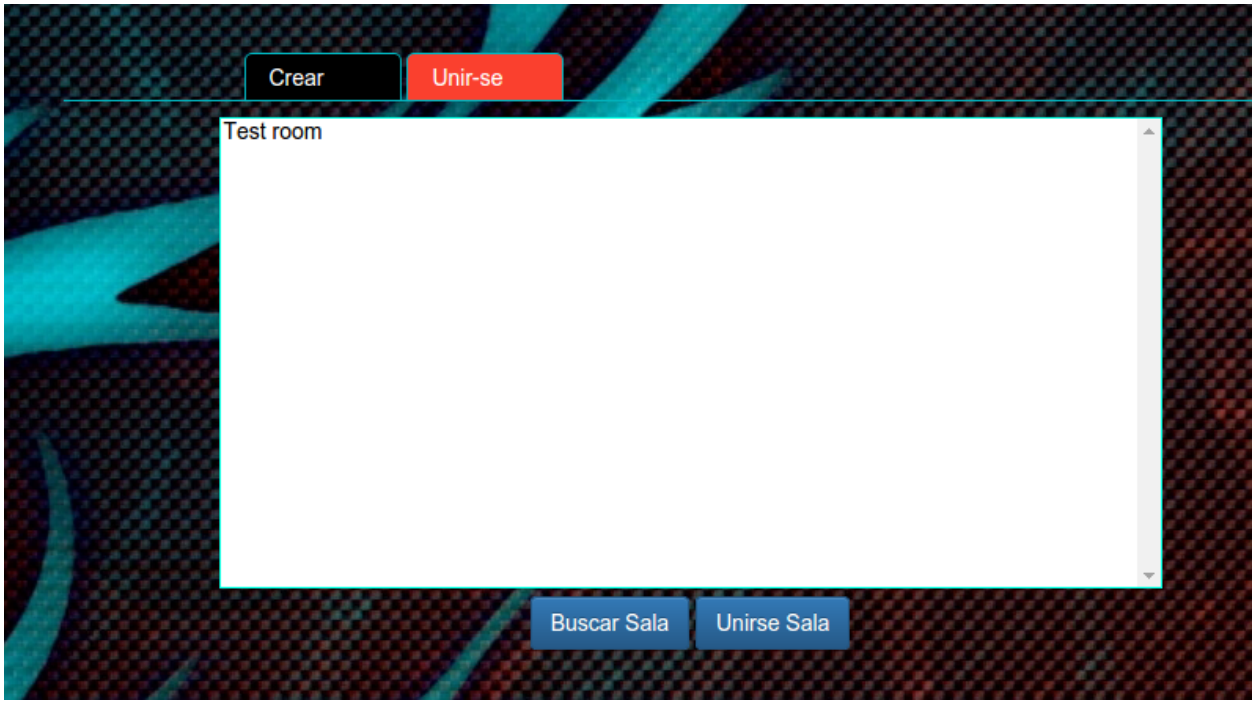

Figura 47. Llistat de les sales a les quals ens podem unir.

Un cop tenim 2 jugadors a la partida, aquesta comença:

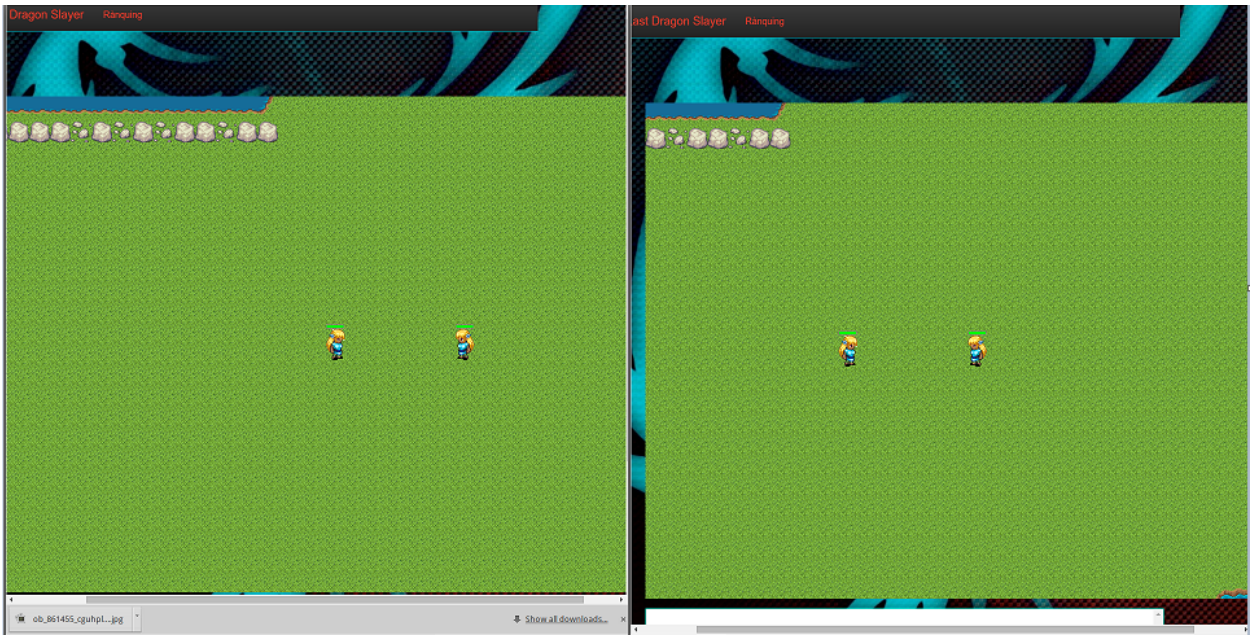

Figura 48. Inici de partida

I podem veure que el personatge és pot moure i es sincronitza amb l'altre client:

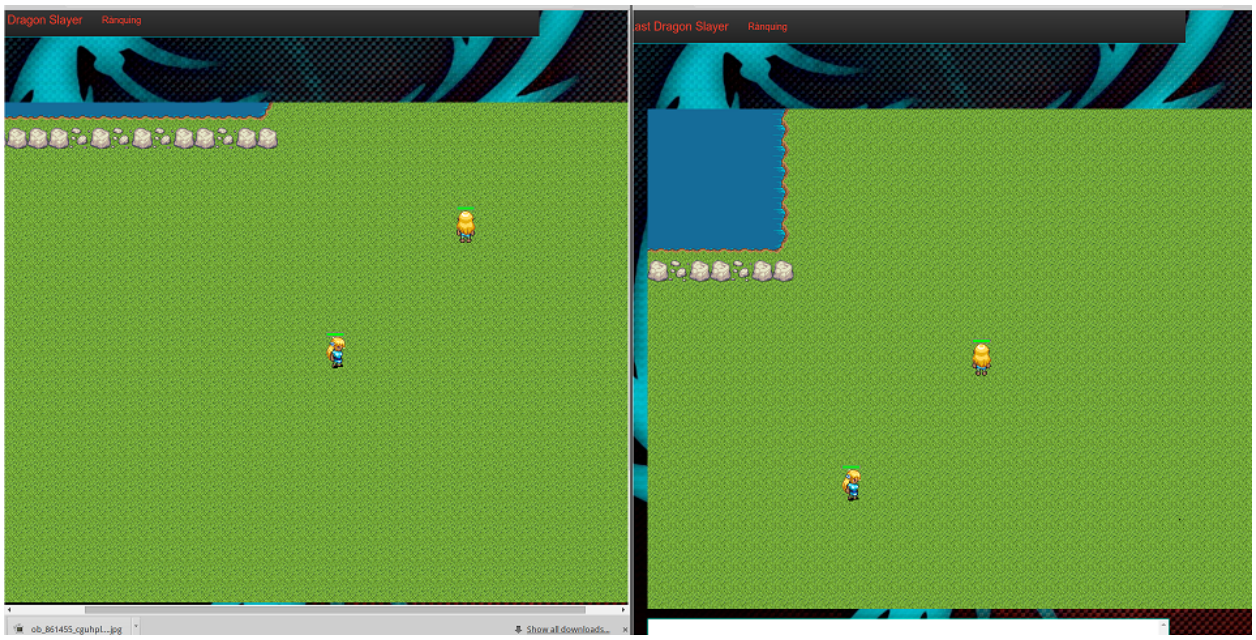

Figura 49. El personatge és mou i queda sincronitzat

També veiem que pot llençar un conjur:

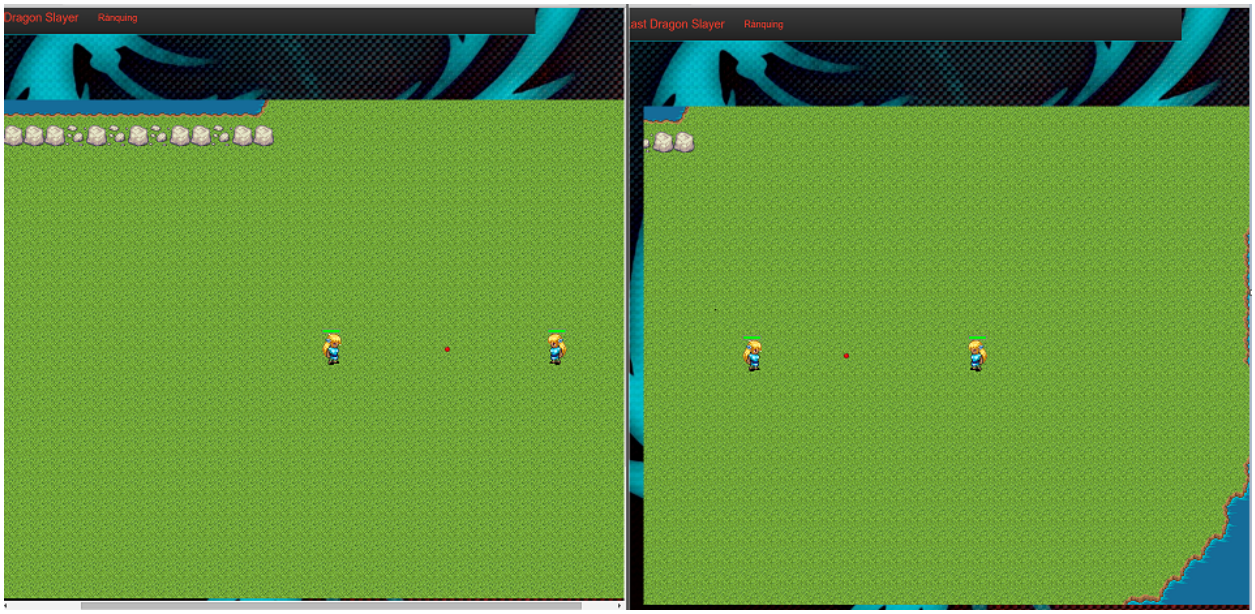

Figura 50. El personatge llença un conjur

Quan la partida acaba és mostra una pantalla de victòria o derrota:

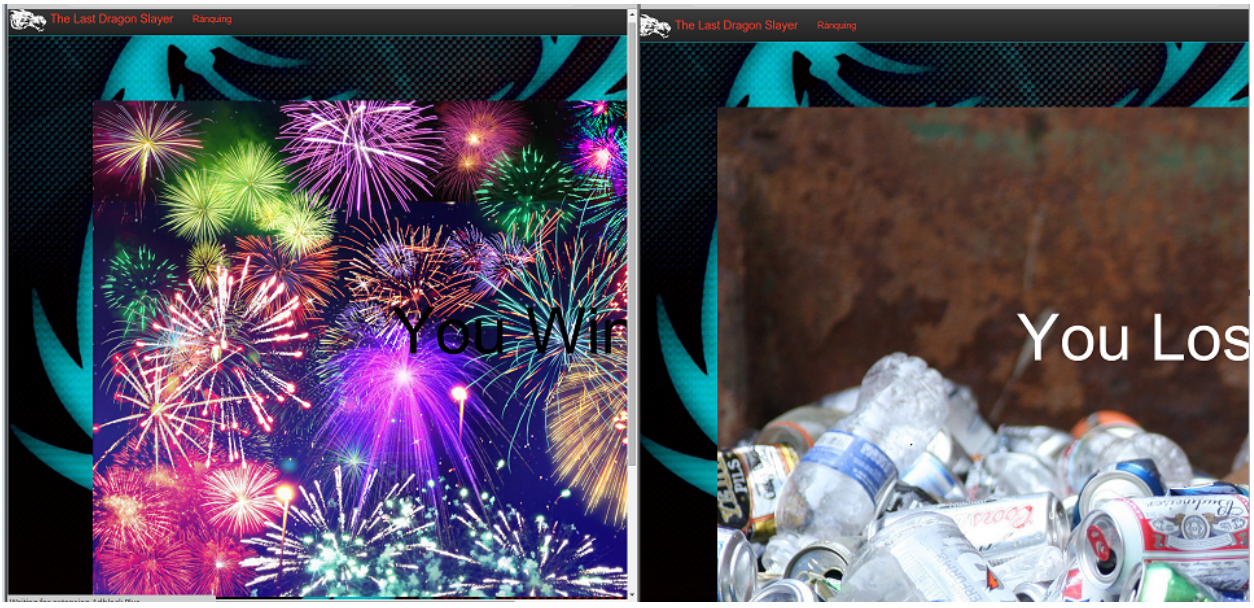

Figura 51. Fi de la partida

Per finalitzar aquest apartat veurem com els jugadors poden intercanviar missatges mentre estan dins d'una partida:

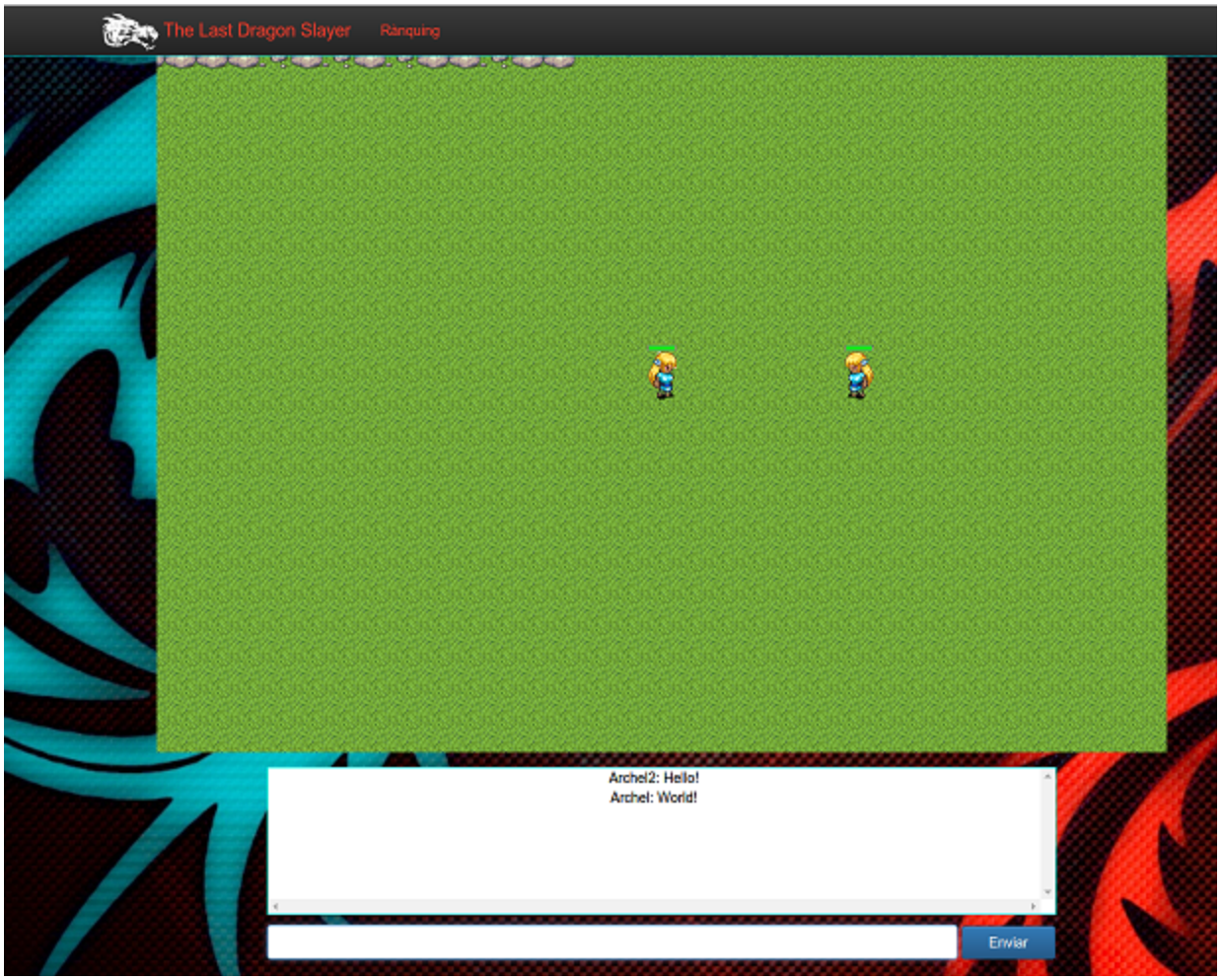

Figura 52. Xat

11. Conclusions

L'objectiu d'aquest projecte era la implementació d'un joc PVP online amb HTML5 i Nodejs.

Els resultats obtinguts són els següents:

1. S'ha programat un joc PVP online, tant part del client com part del servidor on els jugadors poden crear i unir-se a partides per jugar.
2. S'ha dissenyat un web, perquè sigui el més usable i simple possible, ja que el nostre objectiu és que els usuaris juguin.
3. Molt d'aprenentatge ja que les tecnologies emprades per fer el projecte no han estat apreses a l'universitat.

11.1. Opinió personal

Aquest treball m'ha servit per entendre millor el llenguatge emprat en aquest projecte, el javascript.

Ha estat un repte personal, ja que he hagut de compaginar la programació amb la feina. Treure temps per fer aquest projecte ha sigut molt complicat, cosa que m'ha acabat endarrerint bastant.

Tot i així he après molt. M'ha anat molt bé per aprofundir un llenguatge que com a programador web faig servir molt habitualment, però que en utilitzar frameworks com jQuery no feia falta que conegues en profunditat.

Després d'acabar el projecte entenc molt millor el comportament del javascript, així com la creació de pseudo-classes i mòduls. També comprenc molt millor el funcionament del flux asíncron que fa servir.

El més difícil ha estat implementar i dissenyar la part del servidor, ja que les tecnologies emprades eren completament desconegudes per mi. També ha estat un repte programar de manera eficient la part del joc, ja que no tinc gaire experiència en aquest àmbit, però m'agrada molt i estic molt content del resultat obtingut.

12. Treball futur

En aquest punt parlarem de noves funcionalitats que desenvoluparem en un futur.

Les funcionalitats seran les següents:

- Fer que el personatge tingui més atacs.
- Afegir sistema de nivells i recompenses en pujar-hi.
- Fer sistema de *matchmaking*[20].
- Crear sistema de IA per poder practicar contra la màquina.
- Fer que els jugadors es puguin afegir a la partida com a espectadors.
- Fer que el login pugui ser amb altres comptes (facebook, twitter, google plus, steam, etc.).
- Afegir nous personatges, tant gratuïts com de pagament.
- Creació d' editor de mapes, perquè els usuaris pugin crear els seus propis.

[20] Matchmaking: Sistema per agrupar jugadors (2 o més) basat en algunes característiques, per no fer que els jugadors novells juguin amb els experts.

13. Bibliografia

1. jQuery. (sense data). Recollit de http://jquery.com

2. Bootstrap. (sense data). Recollit de http://getbootstrap.com

3. Phaser. (sense data). Recollit de https://phaser.io

4. Nodejs. (sense data). Recollit de https://nodejs.org

5. Socket.io. (sense data). Recollit de http://socket.io/

6. js-signals. (sense data). Recollit de http://millermedeiros.github.io/js-signals/

7. Express. (sense data). Recollit de http://expressjs.com

8. Mongodb. (sense data). Recollit de https://www.mongodb.org/

9. Mongoose. (sense data). Recollit de http://mongoosejs.com/

10. Colt McAnlis , Peter Lubbers , Duncan Tebbs , Brandon Jones , Andrzej Mazur , Sean Bennett , Florian d'Erfurth , Bruno Garcia , Shun Lin , Ivan Popelyshev , Jason Gauci , Jon Howard , Ian Ballantyne , Jesse Freeman , Takuo Kihira , Tyler Smith ,Don Olmstead , John McCutchan , Chad Austin , Mario Andres Pagella (23 / 04 / 2014). HTML 5 Game development insights.

11. TeamGantt. (sense data). Consultat el 06 / 2014, a https://teamgantt.com

12. StackOverflow. (sense data). Recollit de http://stackoverflow.com/

13. Websockets. (sense data). Recollit de https://developer.mozilla.org/es/docs/WebSockets-840092-dup

14. Webworkers. (sense data). Recollit de http://www.w3schools.com/html/html5_webworkers.asp

15. Canvas. (sense data). Recollit de http://www.w3schools.com/html/html5_canvas.asp

16. WebGL. (sense data). Recollit de https://developer.mozilla.org/es/docs/Web/WebGL

17. Concurrency model and event loop. (sense data). Recollit de https://developer.mozilla.org/es/docs/Web/JavaScript/EventLoop

18. Redis. (sense data). Recollit de http://redis.io/

19. Cassandra. (sense data). Recollit de http://cassandra.apache.org/

20. Mongodb. (sense data). Recollit de https://www.mongodb.org/

21. Neo4j. (sense data). Recollit de http://neo4j.com/

22. Tiled map editor. (sense data). Recollit de http://www.mapeditor.org/

23. Shoebox. (sense data). Recollit de http://renderhjs.net/shoebox/

24. Impactjs. (sense data). Recollit de http://impactjs.com/

25. Kiwijs. (sense data). Recollit de http://www.kiwijs.org/

26. Melonjs. (sense data). Recollit de http://melonjs.org/

27. Gamejs. (sense data). Recollit de http://gamejs.org/

28. Phaserjs. (sense data). Recollit de https://phaser.io/

29. Pixijs. (sense data). Recollit de http://www.pixijs.com/

30. Benchmarking Node.js - basic performance tests against Apache + PHP. (sense data). Recollit de http://zgadzaj.com/benchmarking-nodejs-basic-performance-tests-against-apache-php

31. Expressjs. (sense data). Recollit de http://expressjs.com/es/

32. Moongose. (sense data). Recollit de http://mongoosejs.com/

33. Sublime text. (sense data). Recollit de http://www.sublimetext.com/3

34. Git. (sense data). Recollit de https://git-scm.com/

35. Subversion. (sense data). Recollit de https://subversion.apache.org/

36. Tortoisesvn. (sense data). Recollit de http://tortoisesvn.net/

37. Conceptos sobre APIs REST. (sense data). Recollit de http://asiermarques.com/2013/conceptos-sobre-apis-rest/

38. Function.prototype.call. (sense data). Recollit de https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Function/call

39. NPM. (sense data). Recollit de https://www.npmjs.com/