CRASH-ME

JOC EN HTML5

MEMÒRIA PROJECTE FINAL D'ESTUDIS

VÍCTOR RUBIELLA MONFORT

ROMÁN ROSET MAYALS

UNIVERSITAT OBERTA DE CATALUNYA

Índex

Agraïments

Dedico aquest treball molt especialment al meu petit de 2 anys, al que durant aquests últims mesos no li he pogut dedicar el temps que es mereix, el seu tarannà sempre alegre i feliç és la meva principal inspiració.

A la meva dona, sense la seva comprensió i suport mai hagués pogut finalitzar aquesta carrera i menys aquest projecte.

Agraeixo molt a la UOC i a tots els professors i consultors per donar-me el suport i medis necessaris per poder assolir aquesta fita amb èxit.

Agraïments especials a en Román, el meu consultor, pel seu suport i donar-me llum quan més ho necessitava!

Prefaci

HTML5 i Javascript estan de moda. No són precisament tecnologies noves, les dues van néixer als anys 90. Han rebut però el suport d'importants companyies, sobretot pel seu caràcter multiplataforma, en un mercat boig i saturat de dispositius mòbils de tota mena. Les millores aportades en les últimes revisions han fet que tornin a situar-se a primera línia de foc.

Com a desenvolupador mai he fet un ús intensiu d'aquestes tecnologies, doncs els meus orígens es van focalitzar en aplicacions d'escriptori tradicionals (C++,V. Basic, .Net, Java,...). Actualment, m'he especialitzat en arquitectures web (J2EE, PHP, ...), però sempre del costat de la programació del servidor, així i tot com és lògic tinc un coneixement bàsic de Javascript i HTML implícits en aquestes arquitectures.

Amb una contínua motivació per aprendre, el meu objectiu a curt termini se centra en els dispositius mòbils. He considerat que aquestes tecnologies són la millor forma d'introduir-me en aquest nou món tan heterogeni i canviant, sobretot per ser actualment les més estandaritzades.

Introducció

El projecte a realitzar consisteix en un videojoc de tipus puzzle competitiu, que ha de poder executar-se en navegadors web i sistemes operatius Android.

Aquesta memòria té un caràcter clarament experimental i formatiu, per aquest motiu no es pretén desenvolupar un joc molt elaborat i complex, sinó presenciar les dificultats i reflexions, especialment des del punt de vista tecnològic que comporten el desenvolupament de jocs per a dispositius mòbils.

Per tal de fer la memòria més clara i concreta pressuposaré al lector uns coneixements previs com a desenvolupador mínimament familiaritzat amb tecnologies i terminologia web i em centraré específicament en els nous coneixements adquirits. Així i tot, s'inclou una secció de glossari al final d'aquesta memòria.

Calendari d'Entregues

Mostrem una planificació inicial del projecte. El caràcter clarament formatiu d'aquest projecte i memòria pot fer que la distribució de tasques es modifiqui al no disposar del coneixement suficient per fer una estimació acurada. Per aquest motiu, definirem tota una sèrie de funcionalitats bàsiques i opcionals.

	Tasques	Estimació	Lliurament
Fase 1	Planificació	1 dia	
	Definició del Joc	1 dia	
	Formació Programació de Videojocs	1 dia	
	Investigació i selecció de frameworks i APIs	2 dies	
	"Spike" o Petita prova de la tecnologia	7 dia	
			13/03/2013
Fase 2	Joc individual bàsic	14 dies	
	Anàlisis Desplegament	7 dies	
	Documentació	2 dies	
			10/04/2013
Fase 3	Documentació anàlisi desplegament	7 dies	
	Investigació motor renderitzat	7 dies	
	Proves i correccions a diferents *tablets*	7 dies	
		7 dies	**08/05/2013**
Fase 4 -	Optimització i Extres.	14 dies	
Entrega	Vídeo Demostració del videojoc	1 dia	
Final	Preparar Presentació i fer vídeo	7 dies	
	Revisió final i entrega	7 dies	
			05/06/2012

Definició del Joc

Descripció General

El Joc consisteix en anar combinant blocs de colors dins un espai limitat. Els blocs apareixen en la part inferior (terra) i es mouen verticalment cap a la part superior (sostre). El joc finalitza quan un bloc toca el sostre. Guanya el jugador amb més puntuació.

La puntuació s'obté combinant blocs i aquesta depèn de diferents factors: el grau de complexitat de la combinació, el nivell de dificultat del joc o el número de blocs que intervenen.

Per poder realitzar les combinacions, el jugador pot anar intercanviant la posició dels blocs de forma horitzontal.

Tenim blocs de diferents colors i tipus, i no tots reaccionen igual a una combinació, uns desapareixen, altres es transformen, etc. Normalment, els blocs no poden flotar, per tant, sempre cauran fins arribar al terra o trobar un altre bloc, d'aquesta manera evitem que arribin al sostre.

Quan juguem contra un altre jugador, certes combinacions de blocs produiran atacs sobre el nostre rival, enviant-li blocs especials dins el seu espai de joc.

Especificació

Requeriments Bàsics.

- ✓ Mode de joc individual
- ✓ Disposem de 6 blocs diferents que aniran apareixent de forma aleatòria per la part inferior.
- ✓ Podem veure quina serà la següent seqüència de blocs que apareixerà.
- ✓ Es detecten correctament totes les combinacions possibles, els blocs desapareixen.
- ✓ Quan es produeix una combinació per l'efecte de caiguda d'un bloc, multiplicarà la puntuació de la combinació original i així successivament.

- ✓ La velocitat de pujada dels blocs depèn del nivell de dificultat
- ✓ S'ha de mostrar un marcador amb la puntuació i nivell actuals.
- ✓ Disposem d'un menú d'opcions abans d'iniciar la partida.
- ✓ Quan produïm combinacions ens apareixeran blocs especials que cauran de sobre. Els anomenarem pedres.
- ✓ Si produïm una combinació adjacent a una pedra, aquesta es trenca i es transforma en blocs normals.

Requeriments Desitjables

- ✓ Podem tenir pedres de diferents materials, segons el material poden necessitar més o menys impactes per trencar-se.
- ✓ Afegir música i so ambient.
- ✓ Afegir animacions especials quan es produeixen combinacions, quan queda poc perquè algú toqui el sostre, etc..
- ✓ Disposar d'un menú amb opcions que es pugui configurar.
- ✓ Mode 1vIA.
- ✓ Diferents disposicions inicials dels blocs segons la dificultat.
- ✓ Blocs especials que al combinar-los generen blocs normals.

Requeriments addicionals

- ✓ Mode 1v1 online.
- ✓ Registre de jugadors.
- ✓ Rànquing de puntuacions.
- ✓ Tutorial.
- ✓ Xat.

Selecció Tecnologia (Frameworks i APIs)

Ja hem comentat que el joc es realitzarà en Javascript i HTML5 i que aquestes tecnologies estan de moda, per tant, només hem de fer una cerca a internet per veure la gran quantitat de frameworks, API's, biblioteques, ... que hi ha disponibles.

L'excés d'informació pot ser tan dolent com la manca. El desconeixement i el fet de disposar de tantíssims recursos i poc temps per decidir-nos, pot resultar una mica estressant. Per facilitar-nos aquesta tasca hem de fer una bona reflexió sobre què necessitem realment, intentant buscar alguna font fiable que faci alguna comparació entre diferents alternatives i, finalment, aplicar el nostre sentit comú en base a les nostres necessitats.

Què necessito

Tenim frameworks i APIs de tot tipus i per a tot, però en el meu cas he focalitzat la meva atenció en els que treballen amb funcionalitats concretes dels videojocs, com la renderització, la gestió multimèdia,... no tant en l'arquitectura, la qual prefereixo gestionar i definir manualment.

Tot seguit definiré quines han estat les meves fites i per què:

- ✓ Compatible amb events mòbils (Touch, multitouch,etc.).
- ✓ Portable a codi natiu (Android o iOS).
- ✓ Eficient.
- ✓ Projecte actiu i ben documentat.
- ✓ De baix nivell: senzilla d'utilitzar i flexible.
- ✓ Orientada a videojocs.

El fet que sigui compatible amb events mòbils o que sigui fàcilment portable a codi natiu i orientada a videojocs, no mereix gaire explicació ja que forma part dels objectius d'aquest projecte.

Vull que sigui eficient perquè els dispositius mòbils tenen una capacitat de procés molt més limitada que un ordinador de sobretaula, a més tractant-se d'un joc senzill en 2d.

Ha de ser un projecte actiu i ben documentat perquè no disposo de gaire temps de formació i necessito rendibilitzar al màxim aquesta formació. A més, vull que sigui profitosa en un futur, sobretot en un mercat molt actiu i canviant, és necessari que el projecte estigui constantment actualitzant-se per adaptar-se al mercat i realment m'aporti un gran valor.

De baix nivell i flexible, perquè no vull que em predetermini una forma de treballar, estic aprenent i vull entendre com funciona per dins, disposar del control total quan ho cregui oportú per experimentar. També és important aquest control en una fase final d'optimització.

Decisió

Després de donar un cop d'ull molt ràpid a diferents alternatives com JQuery Game, CraftyJS o LimeJS em vaig centrar en dos: KineticJS i CreativeJS.

CreativeJS de fet és una "suite" que engloba diferents APIS Javascript, les competidores amb KineticJS serien en aquest cas EaselJS i TweenJS, una per treballar amb canvas i l'altra especialitzada en realitzar animacions. A més, aquesta suite compta amb el suport oficial de Adobe i diferents plugins per treballar amb flash.

Tot i les aparentment excel·lents i completes funcionalitats que ofereix EaselJS, finalment m'he decantat per KineticJS. Per una banda, Easel no està dissenyada ni optimitzada per treballar amb dispositius mòbils i la seva arquitectura és força més complexa i pesada. També he pogut llegir en els fòrums que alguns usuaris han experimentat problemes de rendiment a l'hora de fer la portabilitat a dispositius mòbils.

KineticJS per la seva banda és una API senzilla i lleugera, amb un codi ben organitzat i fàcil d'entendre, amb una documentació senzilla, però útil i amb molts exemples aclaridors. Ofereix un conjunt de funcionalitats suficient com les capes, filtres, animacions, ..., a demés em permet treballar molt fàcilment a baix nivell accedint a l'objecte canvas. Disposa d'un sistema d'events compatible amb mòbils. Per acabar de decidir-me, sembla disposar de desenvolupadors molt actius, doncs l'actualitzen 2 o 3 cops cada mes, corregint bugs i afegint

funcionalitats, això és molt important ja que em dóna la sensació que si em sorgeix qualsevol problema o bug durant la portabilitat, podré reportar-ho i rebré una resposta satisfactòria.

Finalment i com analitzarem més endavant, no he utilitzat cap framework. Encara que vaig començar el desenvolupament amb KineticJS, els problemes de rendiment que vaig tenir inicialment em van dur a tornar a programar manualment el renderitzador.

Portabilitat

El codi HTML 5 i Javascript per si mateix ja es podria executar en qualsevol navegador modern de qualsevol dispositiu, però al tractar-se d'un joc on es fa un ús intensiu de continguts multimèdia, el seu rendiment pot veure's força afectat en dispositius que no siguin molt potents.

Precisament a causa d'aquest aspecte, que no vaig valorar degudament a l'inici del projecte, he sofert greus problemes de rendiment al desplegar el joc sobre dispositius de gama mitja-baixa. Donat el caràcter formatiu del projecte, he decidit dedicar-hi esforços a analitzar a fons les opcions que hi ha per millorar aquest rendiment en comptes de desenvolupar funcionalment un joc més complex.

Formació Específica en Videojocs

Desenvolupar un videojoc té poc a veure amb fer una aplicació d'escriptori o una aplicació web, així doncs el primer que hauríem de fer a l'hora de voler dissenyar l'arquitectura que utilitzarem i el model de dades és documentar-nos una mica sobre com es construeix un videojoc, patrons utilitzats,... com a mínim per poder disposar d'unes bases amb garanties d'èxit.

La principal característica que podem veure a l'hora de realitzar un joc és que tot el nostre codi es trobarà inclòs dins un bucle infinit, conegut com a "game loop", que iterarà fins que sortim del joc.

Cada iteració del bucle comporta una renderització de tots els components i, per tant, si ho fem de forma sincronitzada per a que cada inici del bucle es produeixi cada cert temps, obtenim el que anomenem FPS (*frames per second*).

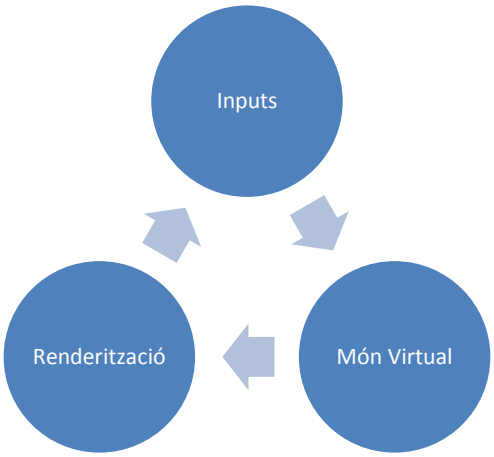
Tenim variants a aquesta idea base, com crear bucles amb freqüències diferents per gestionar certs recursos de forma independent. Nosaltres d'entrada no ens complicarem i utilitzarem com a model bàsic.

Una altra característica és que un joc està contínuament actualitzant el seu món virtual encara que l´usuari no interactuï directament amb ell, podríem dir que un joc és una aplicació "viva" amb una "pseudo-consciència" pròpia.

Game Loop

Un "Game Loop" bàsic es divideix en 3 fases:

1. Interpretar events d'entrada (interacció amb l'usuari).
2. Actualitzar el món virtual.
3. Renderitzar una part del món virtual.



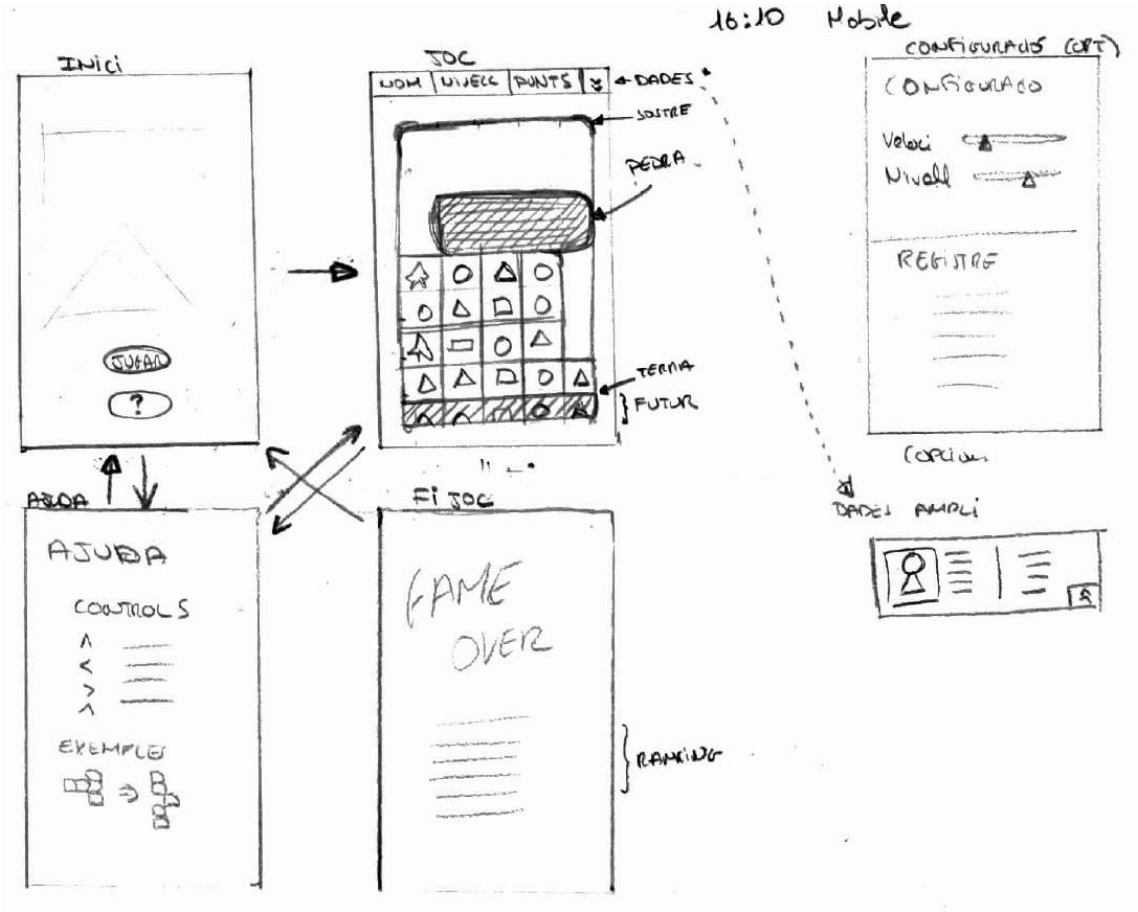
Com ja hem comentat, el bucle itera infinitament cada cert temps, per tant, el temps de procés d'aquestes fases ha de ser inferior a la velocitat d'iteració sinó obtindrem comportaments inesperats.

Desenvolupament

Disseny

Sketch

Començarem amb un sketch a mà alçada del que volem fer:



Podem distingir les principals pantalles que compondran el joc i veure el flux de navegació entre elles. Inicialment, ens centrarem en tres: Inici, Joc i Fi Joc.

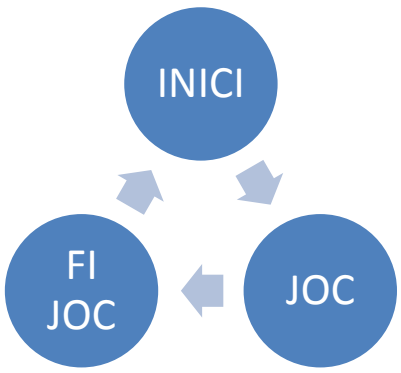
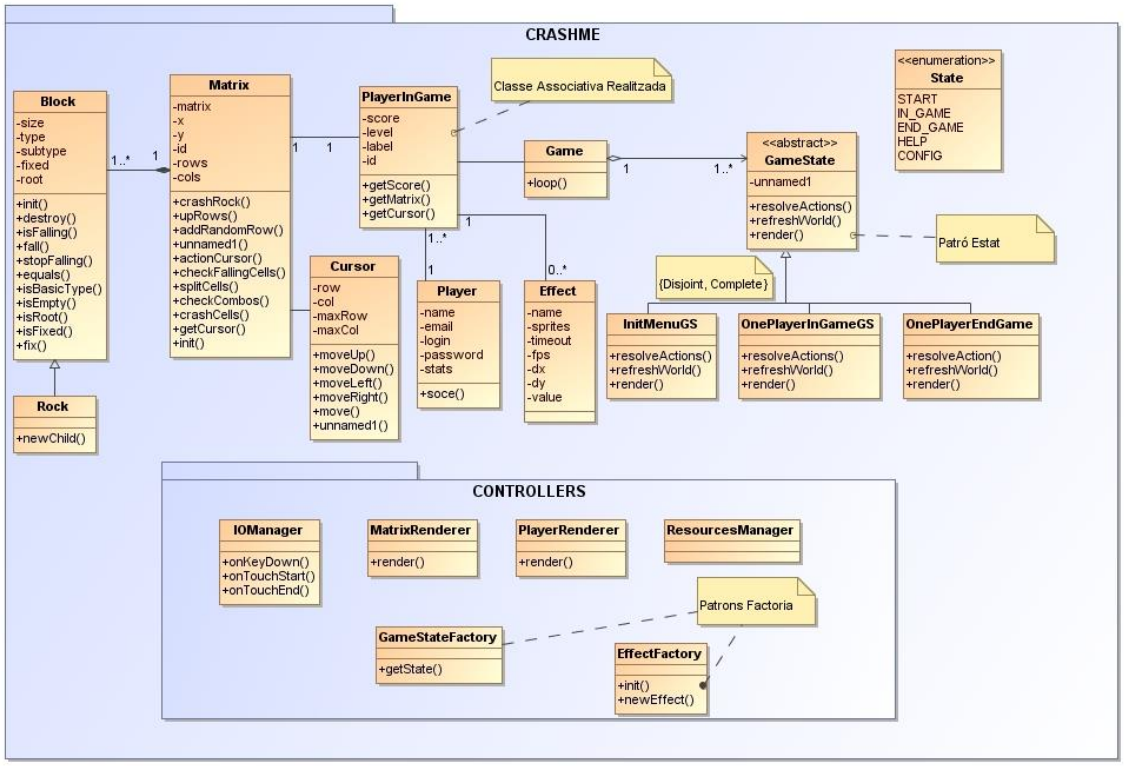


Diagrama de classes

A continuació, dissenyem el diagrama de classes que utilitzarem:



Disposem de la classe Game que contindrà el bucle principal del joc: loop game. Com que cada pantalla disposarà d'una lògica e interacció diferents, podem considerar que el joc estarà en estats diferents i aplicarem el patró estat: GameState.

Podem trobar exemples de codis font on aquests estats han sigut representats amb un "switch". Però encara que a priori sembli una complicació addicional, el patró estat ens donarà molta flexibilitat per afegir noves pantalles mantenint el codi clar i net.

Per crear fàcilment aquests estats farem ús del patró factoria que és implementat per GameStateFactory. GameStateFactory a més de construir els diferents estats ens farà de "pool" de manera que no sigui necessari estar generant estats nous en temps d'execució. Més endavant ho analitzarem amb més detall.

PlayerInGame representa un jugador i guardarà les dades generals d'aquest, com la puntuació el nivell de dificultat, etc.

Cada jugador disposa d'una matriu plena de blocs de colors que són representades per Matrix i Block. Com que tenim blocs especials que tenen lògica específica, les pedres, disposem d'una especialització de la classe Bloc: Rock. La classe Matrix, a més de contenir una matriu de blocs conté la lògica de negoci necessària per manipular tots aquests blocs dins la matriu.

Finalment, tenim 3 controladors. Un per gestionar la interacció amb l'usuari: IOManager, qui controlarà tots els events que l'usuari produeixi. Dos per renderitzar els models a la vista: MatrixRender i PlayerRender.

Configuració entorn de treball

Desenvoluparé en local en un entorn Windows 7 amb l'IDE Aptana Studio 3. Com a repositori de versions utilitzaré Subversion. El mateix Subversion l'utilizaré com a eina per desplegar el projecte en un servidor web (Apache).

Respecte els directoris, el projecte es divideix en 5 directoris principals:

- **css**: conté les fulles d'estils.
- **img**: conté els assets i recursos visuals necessaris.
- **src:** Codi font.
    - **client:** Scripts que s'executaran en client.
    - **server:** Scriprs que s'executaran al servidor.
- **vendor:** Biblioteques externes.

Programació - Plantejament

Abans d'analitzar el codi font, veurem com he pensat representar la matriu de blocs i la interacció del jugador des d'un punt de vista més teòric.

Columna de blocs

Cada jugador disposa d'un espai delimitat de joc en el qual aniran apareixent blocs a la part inferior de la pantalla, el jugador podrà moure aquests blocs de forma horitzontal. A més, haurem de calcular combinacions possibles tant en vertical com en horitzontal. Per aquests motius, la forma més senzilla de representar aquestes columnes de blocs és amb una matriu bidimensional.

Originalment podem pensar en una matriu numèrica de NxN cel·les, on el número de cada cel·la representa un tipus de bloc en concret i si apareix un 0 voldrà dir que està buida:

0	0	0	0	0
0	0	0	0	0
1	0	0	3	2
2	1	0	2	4
3	2	0	2	1
1	2	3	1	3

Aquesta representació ens facilitarà dos tasques: renderització dels blocs, on per representar-los visualment, només necessitarem saber la posició, la mida i el color que utilitzarem (identificat pel número) i, d'altra banda, la gestió de col·lisions o detecció de combinacions.

Detecció de col·lisions

Els blocs per defecte no poden flotar, és a dir, s'han de recolzar sobre altres blocs o sobre el terra, d'aquesta forma el detector de col·lisions per gravetat consisteix en revisar si hi ha alguna cel·la buida a sota d'un bloc.

Veiem el següent exemple:

0	0	0	0	0
0	0	0	0	0
1	0	0	3	2
2	1	0	2	4
0	2	0	2	1
4	2	3	1	3

Quan ens trobem en aquesta situació, els blocs 1 i 2 haurien de caure una posició. Per aconseguir aquest efecte haurem de recórrer tota la matriu a la cerca de blocs que compleixin aquestes característiques. Encara que és una operació a priori senzilla, hem de tenir en compte un detall important: la matriu s'ha de recórrer d'abaix cap a d'alt.

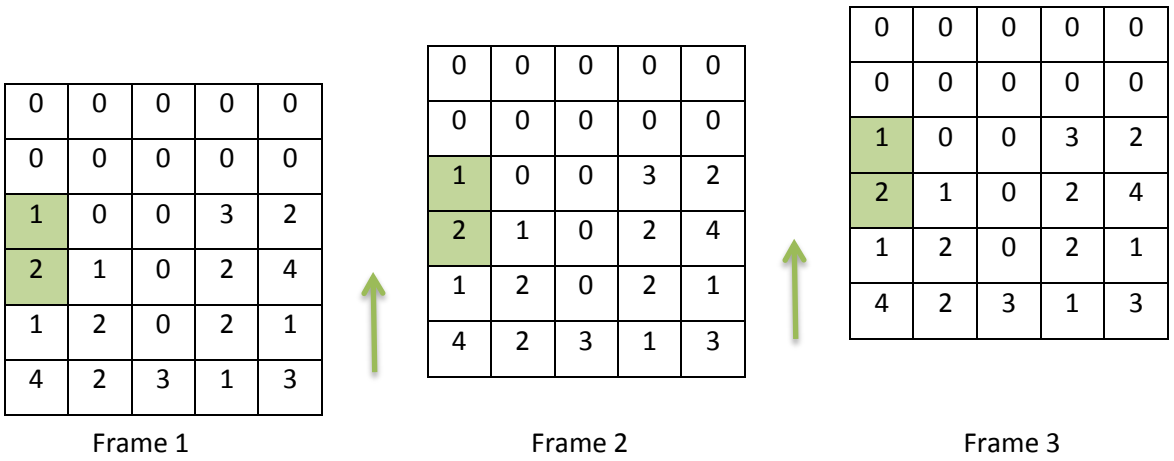
Si recorrem la matriu de d'alt cap a baix, en l'exemple que hem exposat, el bloc 1 no cauria, només cauria el bloc 2, perquè quan estem analitzant el bloc 1 aquest té un bloc sota seu que l'impedeix caure que serà el 2, encara que aquest després caigui, ja no tornarem a analitzar el bloc 1 !.

Moviment de la matriu

Els blocs estan en constant moviment, o s'eleven arrossegats pels blocs que van apareixent en la part inferior o cauen. Per representar aquests moviment s'han de valorar diferents factors tant visuals com lògics.

El moviment de caure pot ser ràpid i constant de forma que una forma fàcil de implementar-ho seria simplement intercanviant les posicions dels blocs dins la matriu (com ja hem vist analitzat en l'exemple anterior).

Aquesta solució, però no la podem utilitzar per fer pujar els blocs, per una banda perquè la velocitat de pujada és variable i principalment lenta i, d'altra, perquè ha de ser suau, no ha de semblar que els blocs donen "salts" cap amunt. Així doncs, per representar el moviment cap amunt, a priori, no modificarem l'estat dels blocs de la matriu, sinó que modificarem la posició de renderització d'aquesta, disminuint-la n píxels en cada iteració (frame).



Frame 1 Frame 2 Frame 3

Moviment constant en zig-zag

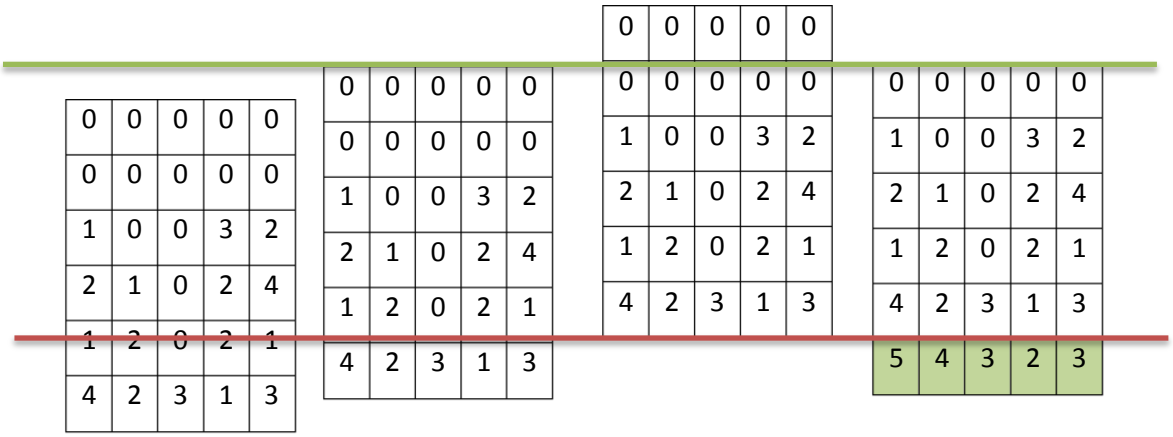
Per crear nous blocs, la solució que sembla més senzilla és afegir una nova fila a la matriu i generar una sèrie de blocs per omplir-la. A la pràctica, això representa diferents problemes: en una matriu clàssica no podem redefinir els seus índexs els quals creixerien indefinidament, si comencem a esborrar índexs la gestió d'aquests es pot tornar complexa i, si no els eliminem, el consum de memòria creixeria exponencialment.

Per resoldre aquest problema he pensat en dos opcions, o bé crear una estructura de dades pròpia amb llistes o mapes amb índexs redefinibles,... o bé aplicar algun truc per reutilitzar els índexs de la matriu i que no creixi.

Disposar d'una matriu simple ens simplifica molt la manipulació de cel·les i la podrem recórrer fàcilment en qualsevol direcció: horitzontal, vertical, diagonal, per patrons, ... si comencem a dissenyar una estructura de dades pròpia, aquests tipus de recorregut podrien tornar-se un mal de cap seriós.

El truc consisteix en què quan vulguem introduir una nova fila, primer movem tots els blocs una posició i afegim la nova fila en l'ultima posició. Per a què el jugador no percebi aquest sotrac, modificarem el punt de renderitzat perquè el moviment visual resultant sigui nul.

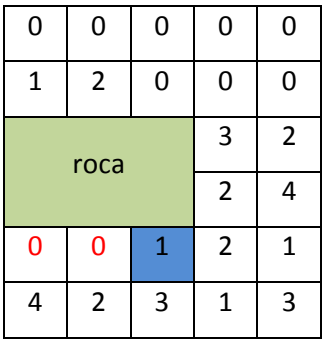
D'aquesta forma la nostra matriu tindrà sempre una mida fixa i constant i s'anirà movent per la pantalla en zig-zag:



Blocs de mida variable

Fins ara hem vist com manipular blocs de mida unitària i fixa, però el nostre joc necessita a més blocs especials, com les pedres que poden ocupar diferents cel·les.

Això que a priori no sembla gaire complex, representa una sèrie de complicacions: La detecció de col·lisions per caiguda ha de tenir en compte que han de estar buides totes les cel·les inferiors de la roca, veiem el següent exemple:



En aquest cas la roca no cau, ja que el bloc 1 s'ho impedeix. Les roques també són fixes, cauen per la gravetat, però el jugador no les pot moure.

També haurem de vigilar al renderitzar, si fins ara recorríem la matriu cel·la a cel·la i renderitzàvem el bloc que teníem dins, amb les roques, no podem fer-ho igual*** .

Per això, introduïm tres nous conceptes: mida de bloc {files, columnes}, bloc arrel (root) i *offset {files,columnes}*. Una pedra, encara que visualment sigui un sol bloc, la representarem com un conjunt de blocs i un d'ells serà el que controlarà a la resta (root). D'aquesta manera, distingirem blocs fills de blocs pares.

Veiem la següent porció de matriu, que representa una roca que ocupa 3 files i 3 columnes (color gris) voltejada per blocs simples de colors.

root mida:{3,3} offfset:{0,0}	child mida:{3,3} offfset:{0,1}	child mida:{3,3} offfset:{0,2}	root {1,1}
child mida:{3,3} offfset:{1,0}	child mida:{3,3} offfset:{1,1}	child mida:{3,3} offfset:{1,2}	root {1,1}
child mida:{3,3} offfset:{2,0}	child mida:{3,3} offfset:{2,1}	child mida:{3,3} offfset:{2,2}	root {1,1}
root {1,1}	root {1,1}	root {1,1}	root {1,1}

Aquesta informació addicional és molt fàcil de generar i ens donarà molts avantatges:

Només analitzarem col·lisions de blocs pare (root) vers altres blocs (poden ser fills o pares).

- La col·lisió simple es manté com fins ara.
- La col·lisió de la roca és fàcil de calcular pel pare, ja que només ha de sumar a la seva posició, la seva mida per saber si té espai per caure o no.
- Quan una es produeix una explosió adjacent a la roca, aquesta ha d'explotar. Si un fill detecta que s'ha produït una col·lisió, pot comunicar-se fàcilment amb el pare a través de *l'offsset* per fer explotar la roca sencera.
- És una solució genèrica que podem aplicar en un futur per blocs especials més grans d'una cel·la.

- El motor de renderitzat només ha de pintar blocs pare segons la seva mida.

El fet d'introduir aquestes noves variables i veure la necessitat de diferenciar certes característiques per cada bloc, com ser inamovibles, ens porta a representar aquestes cel·les com un objecte (classe) propi. Per tant, passarem a tenir una matriu bidimensional d'objectes. Arribats a aquest punt, ens reforça la idea d'estalviar memòria amb la tècnica del zig-zag de la matriu, ja que ara manipularem objectes amb una certa complexitat.

*** Segons l'estratègia de renderitzat i representació de la roca, sí que podríem continuar fent-ho igual, si per exemple representem la roca amb la tècnica de les "tiles" i associem números a parts de roca (costats, centres), etc. Això però, ens limita un mica la part artística i ens complica la gestió de cel·les que componen la roca.

Detecció de combinacions

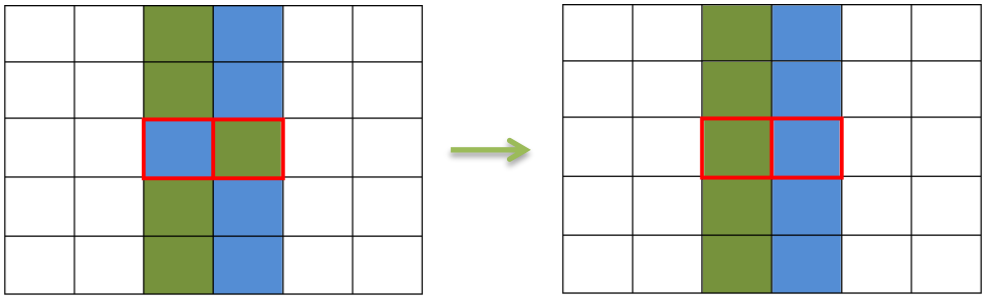
Una combinació es produeix quan ajuntem horitzontal o verticalment 3 o més blocs del mateix tipus. El número màxim de blocs que es poden manipular en una mateixa acció són 2, ja que intercanviem les seves posicions, per tant una acció pot produir com a màxim dos combinacions de forma directa, però moltes de forma indirecta. Si la suma de blocs dins una una o dos combinacions directes és superior a 3, parlem de combinacions especials.

Una combinació indirecta es produeix quan els blocs s'ajunten per l'efecte de la gravetat (caiguda) i no necessàriament per l'acció directa del jugador.

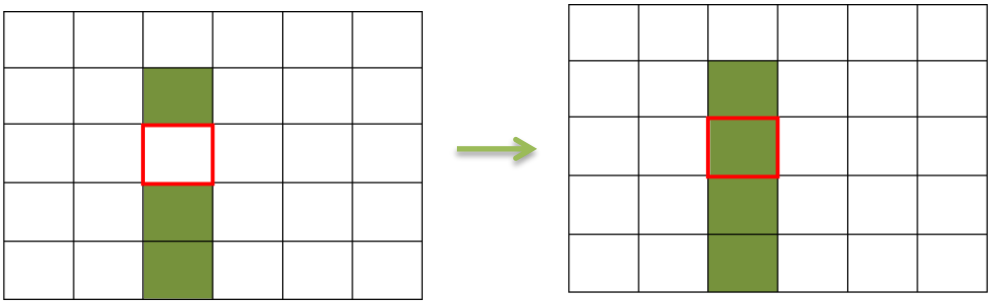
Tant les combinacions directes com les especials tenen una propietat addicional: es poden concatenar i multiplicar el seu valor.

Només es poden produir combinacions quan hi ha canvi de posició de blocs, per tant no és necessari estar recorrent constantment la matriu buscant combinacions. Identifiquem en quins moments es pot produir una combinació:

1. El jugador la provoca al intercanviar la posició de dos blocs. En aquest cas, només cal analitzar els 2 blocs que s'han mogut, prenent com a origen les seves posicions i buscant de forma horitzontal i vertical dins la matriu:



2. Un bloc al caure sobre altres blocs del mateix tipus. Només s'ha d'analitzar el bloc que cau:



3. A l'aparèixer una nova fila de blocs. Tenim que analitzar els blocs de tota la fila.
4. Explosió d'una roca o bloc especial. A l'explosionar les roques o blocs especials es transformen en blocs simples, haurem de recórrer tots aquests nous blocs al igual que ho hem fet al recórrer la fila sencera.

El nostre detector de col·lisions haurà de ser capaç d'apilar les combinacions concatenables de forma que, per una banda mostrem un efecte visual especial, amb un so especial, i altra la puntuació que rebi el jugador sigui més gran.

Renderitzat

Donat que ha estat un punt crític el desplegar el joc en aplicacions mòbils i encara que originalment utilitzava KineticJS, finalment he decidit implementar el meu propi motor de renderitzat per poder optimitzar aquest procés.

Aprofitant que per la lògica del joc he utilitzat una matriu, he agafat la idea base del "Tile Based Engine", on aprofitarem aquesta mateixa matriu a mode de mapa per representar-la gràficament.

Per renderitzar utilitzem una única font d'imatge "img" que conté tots els blocs i pedres de colors que necessita en joc, anem copiant trossets d'aquesta imatge dins el buffer de dibuix, d'aquesta manera optimitzem molt els recursos tant de memòria com de procés al no crear mai cap nova instància per cada bloc.

Per aconseguir això, construïm una imatge que anomenem sprite, on afegirem tots els blocs separats per la mateixa distància entre ells (en el nostre cas sense distància) i per calcular el bloc que tenim que dibuixar només mirem el tipus de bloc (número entre 1 i 8) i el multipliquem per la mida del bloc (64), així obtenim sx, que és la coordenada x origen (source-x).

```
var sx = 0 + cell.getType() * 64;
var sy = 0;
```

Com que les coordenades verticals no es modifiquen i els blocs estan enganxats a la part superior de la imatge, aquesta coordenada en l'origen sempre serà 0 (sy =0).

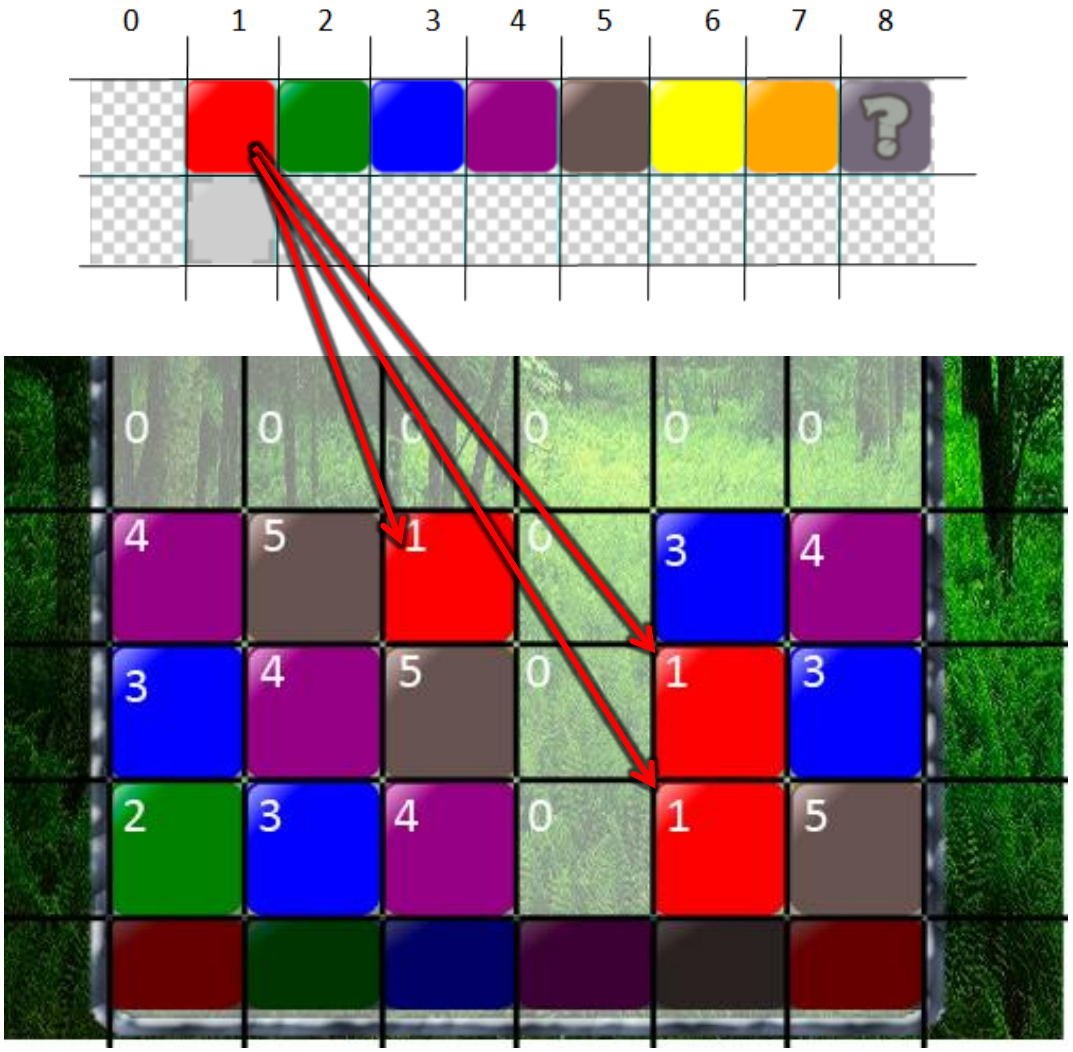
Per calcular les coordenades de destí (dx, dy) partim del "offset" inicial de la matriu. L'Offset inicial són els valors x,y de la matriu que es on comença a dibuixar-se respecte el canvas:

```
var dy = (i *64) + iniY;
var dx = (j *64 ) + iniX;
```

A la posició de la fila i columna que estem dibuixant en aquest moment (i,j) multiplicat per la mida del bloc, si li sumem aquest offset, tenim la posició final.

Podem entendre la idea com anar fent "copiar i enganxar" blocs d'un recurs origen a un recurs final i així anem dibuixant un mapa o matriu, com mostra el següent gràfic:

```
this.ctx.drawImage(this.img,sx,sy,sw,sh,dx,dy,dw,dh);
```



Veiem que aquesta tècnica és força senzilla de gestionar i molt potent. Si en un futur volem afegir, per exemple, efectes de parpadeig, només hem de crear una nova fila de blocs sota els blocs actuals a la imatge font i, a l'hora de renderitzar la matriu, anar alternant l'sy (sy=0, sy =64) sense modificar res més.

Programació - Implementació

Encara que JS és un llenguatge d'scripting no orientat a objectes, o almenys no de forma clàssica amb classes, herència i polimorfisme, ha desenvolupat tècniques pròpies per emular aquests comportaments: prototips. John Resing, expert programador en JS ha desenvolupat una petita i senzilla llibreria (vendor/Class.js) que utilitzant prototips ens permet emular l'herència de classes amb una notació molt similar a la que utilitzem de forma tradicional.

Veiem un exemple d'ús:

```
var GameState = Class.extend({
        init: function(){} //constructor
});

var GameStateMenuIni = GameState.extend({
        init: function(){} //constructor
});
```

A continuació, analitzarem les principals classes i codi desenvolupat, els mètodes mostrats poden no contenir tot el codi font original per donar més claredat a l'explicació.

Game

El nucli principal del joc és el mètode que es troba en aquesta classe:

```
loop: function(){

        this.gameState.resolveActions(this.ioManager,this.players);

        var state = this.gameState.update(this.players);

        if (state != this.gameState.getId()){
                this.gameState = this.gameStateFactory.newGameState(state);
        }

        this.gameState.render();

}
```

resolveActions: Mètode que recupera les accions que s'han de realitzar en aquesta iteració. Aquestes accions són el resultat dels events produïts per l'usuari que ha recollit i interpretat IOManager.

update: Actualització de tota la lògica interna del joc. Durant l'actualització del joc podem adonar-nos que s'ha produït un canvi d'estat, per això Update sempre retornarà un identificador de l'estat on es troba el joc en aquell moment, si aquest és diferent a l'estat actual, l'actualitzarem.

render: Renderització de la pantalla segons l'estat actual del joc.

Com podem veure **loop** és un mètode normal i corrent que no executa cap bucle. Per a que aquest mètode s'executi iteradament de forma sincronitzada amb el temps, utilitzem la funcionalitat global de JS anomenada **setInterval** (/src/client/main.js):

```
var fps = 30;

setInterval(function(){
        game.loop()             //game és una instància de Game
}, 1000/fps);
```

setInterval és una de les funcionalitats clau que ens donarà més problemes a l'hora de portar el nostre joc a dispositius mòbils, és possible fins i tot que haguem de substituir-la per altres mecanismes. Això però, ho analitzarem més endavant.

Podem fer un cop d'ull com s'inicialitza aquest objecte amb els controladors:

```
init: function(){

        this.players = [];

        //controllers
        this.ioManager          = new IOManager();
        this.matrixRenderer     = new MatrixRenderer();
        this.playerRenderer     = new PlayerRenderer();
        this.gameStateFactory   = new GameStateFactory();

        this.state      = Game.START_MENU;
        this.gameState  = this.gameStateFactory.newGameState(Game.START_MENU);

},
```

Una important millora que podríem aplicar a aquest constructor seria la **injecció de dependències**, ja sigui individual o amb un contenidor en comptes d'instanciar dins el constructor directament els controladors, fent un codi difícil de provar, deixem aquesta millora pendent per la següent fase.

GameStateFactory

La implementació clàssica del patró factoria defineix un mapa de factoria que ens permet anar demanant instàncies de classes pel seu nom, seguint un conveni específic. En el nostre cas, implementat una factoria amb pool, de manera que el número d'instàncies que es crearan està acotat.

Això però, no ens evita poder disposar d'estats dinàmics en un futur si ho creiem oportú, de moment, els 3 estats del joc, que representaran les 3 pantalles, els tindrem guardats al mapa de factoria. Veiem com està implementat:

```
init: function(){
```

```
        this.map = {};
        this.map[Game.START_MENU] = new GameStateMenuIni();
        this.map[Game.IN_GAME]    = new GameStateOnePlayer();
        this.map[Game.END_GAME]   = new GameStateGameOver();
},

newGameState: function(id){
        if (this.map[id]==undefined) {
                console.log('ERROR'+ id + 'GameState not found.');
                return null;
        }
        return this.map[id];
}
```

GameStateOnePlayer

GameState principal, doncs és on s'executa tot el joc individual.

```
render: function(matrixR,playerR){
        //Renderitzacions pròpies d'aquest estat.
},

update: function(players){

        var m;
        var nextState = this.getId();

        for(var i=0; i<players.length; i++){
                m= players[i].getMatrix();
                m.update();

                if (m.inTop()){
                        nextState = Game.END_GAME;
                }
        }

        return nextState;
}
```

Veiem que aquest mètode consisteix en recórrer tots els jugadors i actualitzar la seva matriu. Addicionalment, es comprova la condició de fi de joc, **m.inTop** que ens retornarà cert si un jugador té blocs tocant el sostre de la seva matriu.

```
resolveAction: function(action, players){
        switch(action.action){
                case IOManager.SPLIT:
                        cMatrix.actionCellsInPixels(action.sx,action.sy, action.dx, action.dy);
                        break;
                }
        }
}
```

Aquí és on es resolen les accions que ha fet l'usuari. Com veurem més endavant, IOManager transforma els events típics que es poden generar a la vista (click, touch, etc.), en accions de joc. Com exemple mostrem SPLIT que és l'acció que permet a l'usuari intercanviar la posició de dos blocs horitzontalment.

Cell (Block)

La classe Cell representa un bloc dins la matriu d'un jugador. Els blocs normals ocupen una sola cel·la i són de colors, però també tenim blocs especials i pedres, aquestes últimes representades per la classe Rock que extendrà de Cell.

```
init: function(type,size){
        this.type = type; //0 = Empty, lo demas son colores :)
        this.subtype = 0;

        this.size = size;
        this.root = true; //celda de control,
        this.fixed = false;
        this.falling = false;
},
```

Analitzem amb detalls aquests atributs:

- **type**: Tipus de cel·la. Els valors de 1 a 9 representen colors de blocs bàsics. Quan val 0 representa que no hi ha cap bloc, aquí utilitzem el patró NullObject representant per aquest type=0, ens serà molt útil per donar claredat i consistència al codi. Els valors superiors a 9 representen blocs especials, que no tractarem en aquesta fase.
- **subtype:** És el discriminador que utilitzarem per saber si es tracta d'un bloc bàsic o d'una pedra.
- **size:** Mida d'un bloc, per defecte tots els blocs ocupen 1 cel·la.
- **root:** Bloc pare o principal. Si un bloc ocupa una cel·la sempre serà root, però si ocupa diferents cel·les , distingirem un bloc principal i la resta seran fills. Això ho expliquem amb més detall dins de la classe **Rock**.
- **fixed:** Si és cert l'usuari no pot moure aquest bloc, només pot caure per efecte de la gravetat.
- **falling:** Indica si el bloc està caient dins la matriu.

```
Cell.newRandom = function(level){
        var cell = new Cell(Math.floor((Math.random()*(level+5))+1), {rows:1,cols:1});
        return cell;
};

Cell.newEmpty = function(){
        return new Cell(0, {rows:1,cols:1});
};
```

Aquí veiem dos mètodes factoria que utilitzem per generar blocs fàcilment. Un genera el NullObject o bloc buid i l'altre genera un bloc aleatori segons els nivell de dificultat.

En aquest codi també podem veure com representem la mida dels blocs amb una tupla (rows,cols).

Rock

Representa un bloc amb característiques clarament diferenciades del bloc bàsic: sempre és fixe, és a dir, el jugador no el pot moure. Normalment ocupa diferents cel·les i és de mida variable. A més, quan un jugador trenca una pedra, en comptes de desaparèixer es transforma en altres blocs de tipus simple.

Matrix

Aquesta classe representa la matriu de joc d'un jugador i conté tota la funcionalitat necessària per manipular blocs dins la matriu. Veiem primer de tot com s'inicialitza i quins paràmetres té:

```
init: function(id,x,y,rows,cols,player){
        this.x = x;
        this.y = y;
        this.id = id;
        this.rows = rows;
        this.cols = cols;

        this.matrix = [];

        this.player = player;

        this.top = 4; //sostre de la matriu
        this.floor = rows-1;


        this.concatenateValue = 2;

        //Delay timmers
        this.concatenateDelay = 0;
        this.delayFall = 0;
        this.frames = 0;

        this.eFact = new EffectsFactory();

        this.initZeroMatrix(rows,cols);
        this.initLevelMatrix();
},

initMatrixs: [

[[1,2,3,4,5,1],[2,3,4,5,1,2],[3,4,5,1,2,3],[4,5,1,2,3,4]],

[[1,2,3,4,5,6],[2,3,4,5,6,1],[3,4,5,6,1,2],[4,5,6,1,2,3]],

[[1,2,3,0,5,6],[7,1,2,0,4,5],[6,7,1,0,3,4],[5,6,7,0,2,3]],

[[1,0,3,0,5,6],[7,0,2,0,4,5],[6,0,1,0,3,4],[5,0,7,0,2,3]],

[[1,0,3,0,5,6],[7,0,2,0,4,5],[6,0,1,0,3,4],[5,0,7,0,2,3]]
],
```

- **x, y:** Coordenades de la matriu dins el canvas.

- **rows, cols:** número de files i columnes de la matriu.
- **matrix:** array de blocs que conté la matriu.
- **top, floor:** indicadors que limiten l'espai de joc: sostre i terra de l'array de joc. Recordem que a causa de la tècnica del zig-zag descrita anteriorment, aquest valors aniran canviant contínuament, a més l'array sempre serà més gran per la part superior per poder afegir objectes per sobre el cap visual i deixar-los caure, oferint una entrada en escena suau i fàcil d'implementar.
- **frames:** És el contador que utilitzem per controlar el zig-zag.
- **eFact:** Factoria d'efectes. El motiu d'incloure una factoria d'efectes dins la pròpia matriu és perquè molts d'aquests efectes es produiran en localitzacions relatives a posicions de la matriu i d'un inici és on m'ha resultat més senzill de situar, encara però que no m'agrada acoblar aquesta factoria a l'objecte Matriu, en un futur intentaré pensar una millor solució. Un altre aspecte negatiu més fàcil de solucionar en futures versions és injectar la factoria i no crear-a dins el constructor, el que faria la classe més testejable i mantenible.
- **initZeroMatrix:** Omple tota la matriu de blocs buits (Null Objects).
- **initLevelMatrix:** Inicialitza la matriu amb un conjunt de blocs preestablerts segons el nivell de dificultat.
- **initMatrixs:** Conjunt de matrius d'inicialització segons el nivell de dificultat, cada fila correspon a un nivell de dificultat diferent.

Veiem ara els mètodes que inicialitzen la matriu:

```
initZeroMatrix: function(rows,cols){
        this.matrix = new Array(rows);
        for(i=0; i<= this.floor; i++){
                this.matrix[i] = new Array(cols);
                for(j =0; j< this.cols; j++){
                        this.matrix[i][j] = Cell.newEmpty();
                }
        }
},

initLevelMatrix : function(){
        var level = this.player.getLevel();
        var type = 0;
        var matrixPart = this.initMatrixs[level-1];

        for(i=this.floor; i > (this.floor-matrixPart.length); i--){
                for(j =0; j<this.cols; j++){
                        type = matrixPart[this.floor-i][j]

                        this.matrix[i][j] =  new Cell(type,{rows:1,cols:1});

                }
        }
},
```

Veiem ara el mètode que analitza la matriu en busca de combinacions:

```
checkCombos: function(i,j){
        var cell = this.matrix[i][j];
        if (!cell.isBasicType()){
         return {cells: [], comboExtra: false, combo: false};
        }
        if (cell == 0) return {combo:false};

        var comboArrayH = [];
        var comboArrayV = [];
        var comboExtra = false;

        //busqueda horizontal
        var sh = 0; //la propia celda
        for(var jleft = j-1; jleft>=0;jleft--){
                if (this.matrix[i][jleft].isBasicType() && this.matrix[i][jleft].equals(cell)) {
                        sh++;
                        comboArrayH[comboArrayH.length] = {i:i,j:jleft};
                }else{
                        break;
                }
        }
        for(var jrigth = j+1; jrigth<this.matrix[i].length;jrigth++){
                if (this.matrix[i][jrigth].isBasicType() && this.matrix[i][jrigth].equals(cell))
{
                        sh++;
                        comboArrayH[comboArrayH.length] = {i:i,j:jrigth};
                }else{
                        break;
                }
        }
        if (sh<2){
                comboArrayH =[]; //caput!
                sh =0;
        }
        //Vertical
        var sv = 0; //la propia celda
        for(var iup = i-1; iup >= this.top;iup--) {
                if (this.matrix[iup][j].equals(cell)) {
                        sv++;
                        comboArrayV[comboArrayV.length] = {i:iup,j:j};
                }else if (this.matrix[iup][j].isBasicType() &&
                         this.matrix[iup][j].equalsIgnoringFallingStatus(cell) )
                {
                        this.matrix[iup][j].stopFalling();
                        comboArrayV[comboArrayV.length] = {i:iup,j:j};
                        comboExtra = true;
                        sv++;
                }else{
                        break;
                }
        }

        for(var idown = i+1; idown <= this.floor-1; idown++){
                if (this.matrix[idown][j].equals(cell)) {
                        sv++;
                        comboArrayV[comboArrayV.length] = {i:idown,j:j};
                }else{
                        break;
                }
        }

        if(sv<2){
                comboArrayV =[]; //caput!
                sv =0;
        }
        //Aglutinador de blocs combinats
        var scombo = sh + sv +1;

        if(scombo >= 3){
                //la celda actual no esta incluida en las arrays!
                var cells = comboArrayV.concat(comboArrayH);
                cells.push({i:i,j:j});
```

```
                this.crashCells(cells) ;

                return {
                        combo: true,
                        cells: cells,
                        extra: comboExtra
                }
        }else{
                return {
                        combo: false,
                        cells: [],
                        extra:false
                }
        }
},
```

Com podem veure aquest mètode parteix d'un bloc concret i busca primer de forma horitzontal i després de forma vertical blocs iguals que l'original. Veiem que en total són 4 bucles per cada una de les direccions. Els bucles, però finalitzen en quan es troben un bloc diferent de l'original, així que en el fons seran sempre bucles molt petits.

Finalment, si es detecta alguna combinació s'agrupen els blocs afectats i es crida a l'analitzador de combinacions perquè calculi la puntuació pertinent:

```
        parseCombos: function(combos){

                if (combos.combo) {
                        var score = 0;

                        var comboSize = combos.cells.length;
                        var cell = combos.cells[0]; //primera celda para posicionar efectos
                        var ey = (cell.i - this.top) * 64;
                        var ex = cell.j * 64;

                        if (combos.comboExtra || comboSize > 3){
                                if (!this.player.isComboMode()){
                                        this.concatenateValue = 1;
                                        this.player.setComboMode(true);
                                }

                                this.concatenateValue++;

                                score += combos.cells.length * 100 * this.concatenateValue;


                                this.player.addEffect(this.eFact.newEffect('comboExtra',30,2,
                                                ex,ey,"x"+this.concatenateValue + " !"));

                        }else{
                                if(this.player.isComboMode()){
                                        this.player.setComboMode(false);
                                        var rocks = Rock.newComboRocks(this.concatenateValue);
                                        var rock;
                                        var rowPos =0;
                                        var colPos= 0;
                                        var size = 0;
                                        console.log('adding rocks!', rocks);

                                        for(var i = 0; i< rocks.length; i++){
                                                rock = rocks[i];
                                                var size = rock.getSize();
                                                this.addRock(rowPos,Math.round(
                                                Math.random()*(this.cols-size.cols)),rock);
                                                rowPos +=       size.rows;
                                        }
```

```
                        }

                        score += combos.cells.length*100;
                }

                this.player.addEffect(
                        this.eFact.newEffect('addScore',30,10, ex,ey,"+ "+ score));
                this.player.addScore(score);

        }
},
```

L'analitzador de combinacions el que fa és calcular el número de blocs que intervenen en la combinació i calcular la puntuació. També porta el control de les combinacions especials que es concatenen i, per tant, multipliquen la puntuació d'aquestes. Finalment, afegeix efectes de puntuació a la matriu.

Veiem ara com s'afegeix una roca dins la matriu:

```
addRock: function(row,col,rock){
        var size = rock.getSize();
        for (i = 0; i< size.rows; i++ ){
                for (j = 0; j< size.cols; j++ ){
                        if(j+col >= this.cols || i+row >= this.floor){
                                continue;
                        }
                        this.matrix[i+row][j+col] = rock.newChild(i,j);
                }
        }

        this.matrix[row][col].setRoot();
},
```

En aquest mètode veiem la implementació de la tècnica de fills-pares que vaig explicar a la fase 2, ajudant-nos d'un constructor especial "newChild" que crearà un fill de la roca, omplirem la matriu de petits blocs de roca segons la mida d'aquesta.

Podem revisar la resta de mètodes al propi codi font, on entre d'altres tenim com es trenquen les roques o els blocs, com detectar una combinació adjacent a una roca, controlar la caiguda de blocs, etc. Finalitzarem aquesta anàlisi donant-li un cop d'ull al mètode d'actualització de la matriu, que controla l'efecte de zig-zag, la pujada de la matriu i la caiguda de blocs:

```
update: function(){

        this.delayFall++;

        this.frames++;

        if (this.frames % (6- this.player.getLevel()) == 0) {
                this.y--;
        }

        if (this.y  == -64){
                this.y = 0;
                this.upRows();
```

```
            this.addRandomRow();
        }
        //console.log(m.y);

        var res =this.checkFallingCells();
        this.parseCombos(res);
}
```

Veiem que la velocitat de pujada de la matriu (this.y--) és directament proporcional al nivell de dificultat del jugador.

També podem veure que quan this.y ha pujar la mida d'un bloc simple (-64) es fa l'efecte de zig-zag.

Finalment, es controlen les cel·les que cauen o que han de caure i si es genera alguna combinació resultant d'aquesta caiguda (combinacions especials!).

Player

Aquesta classe, la utilitzem per representar un jugador de la partida, aquesta classe s'encarregarà de mantenir la puntuació del jugador, incrementar el nivell de joc segons aquesta puntuació i actualitzar els efectes especials associats al jugador.

```
init: function(id){
        this.id = id;
        this.score      = 0;

        this.effects = [];

        this.comboMode = false;
        this.level = 1;

        this.matrix = new Matrix(this.id,50,0,16,6,this);

},
```

- **score:** puntuació.
- **effects:** llistat d'efectes visuals associats al jugador.
- **comboMode:** el jugador està fent combinacions especials.
- **Level:** nivell de dificultat.
- **Matrix:** objecte Matrix de 16 files per 6 columnes.

```
addScore: function(score){
        this.score += score;

        if (this.level < 5 &&
            this.score > (Math.pow(2,this.level) * 1000))
        {
                this.level++;
        }
```

```
},
```

Cada vegada que s'afegeix una puntuació es comprova si hem d'incrementar el nivell de dificultat del joc, cada vegada necessitarem més puntuació per incrementar de nivell, seguint una funció exponencial.

Tot seguit, veurem com s'actualitzen els efectes visuals, l'únic que hem de fer és recórrer el llistat i comprovar si un efecte ja ha caducat, per esborrar-lo. Els efectes com veurem més endavant, són autònoms pel que fa al funcionament, només hem de tenir cura de crear-los i esborrar-los:

```
updateEffects: function() {
        var len = this.effects.length;
        var i =0
        while ( i< len) {
                if (this.effects[i]  && this.effects[i].isTimeout()){
                        this.effects.splice(i,1);//borra
                }else{
                        i++;
                }
        }
},
```

Effect i EffectFactory

Classe que utilitzem per representar petites animacions autònomes per mostrar puntuacions obtingudes,... El codi d'aquesta classe de moment només contempla el cas de les puntuacions, però l'objectiu és que pugui gestionar sprites d'animacions i poder executar-les en qualsevol moment.

```
newEffect: function (name,fps,timeout,dx,dy,value){
        var img         = this.srcmap[name][1];
        //var sprites = this.srcmap[name][1];
        if (!value) value = name;
        return new Effect(name,img,fps,timeout,dx,dy,value);
}
```

Aquest mètode crea un efecte a les posicions a dx, dy, associant-li un rati de refresc fps i una caducitat (timeout). Aquest timeout és proporcional al seu rati de refresc i no al rati de refresc del videojoc.

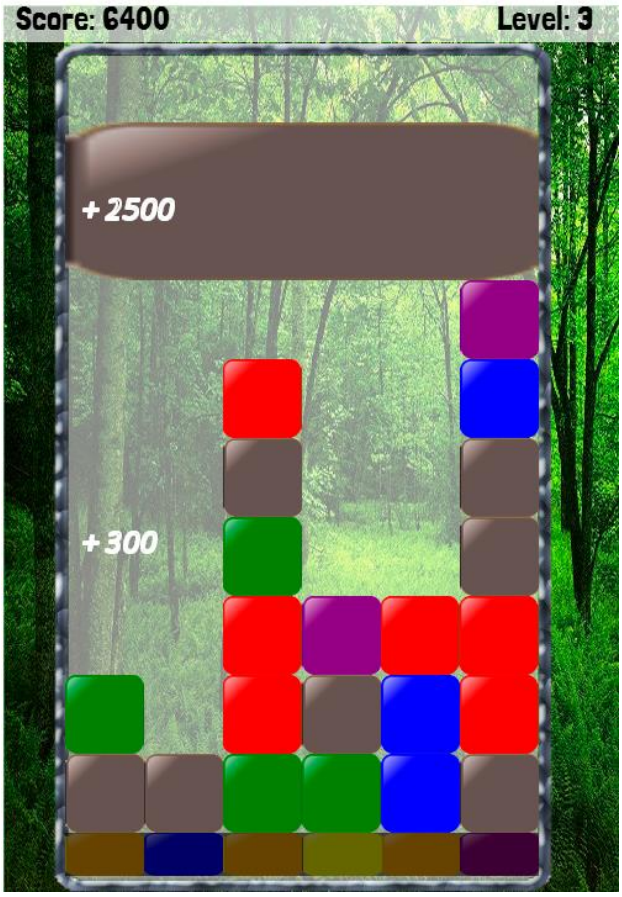
Per entendre com es crea un efecte veiem el nucli del seu constructor:

```
        setInterval(function(){
                if (thiss.name == 'addScore' && thiss.dy > 0) {
                        thiss.dy -=2;
                }
                thiss.i = (thiss.it++ % thiss.sprites.length);

        }, 1000/fps);
```

Efectivament, disposa d'un setInterval propi que li dóna plena autonomia. Veiem que ara mateix només podem crear efectes de tipus "addScore" perquè encara no hem implementat altres tipus, però aquest sistema ens permetria associar imatges amb sprites i refrescar-les segons un interval i així produir animacions independents de la resta del joc.

Tot seguit, mostrem una captura del joc on s'han afegit 2 animacions de puntuació. Aquestes animacions corresponen al +300 i +2500 generades per una combinacions i que aniran pujant per la pantalla fins a desaparèixer pel sostre o transcorreguts uns segons.



IOManager

Encarregada de transformar les interaccions de l'usuari en accions de joc. Captura el events produïts, els interpreta i els afegeix a una pila d'accions per executar.

Dins el bucle principal de joc, cada iteració agafarà aquesta pila d'accions i les executarà una rere l'altre fins buidar la pila.

Veiem primer com associem els events perquè siguin capturats per la nostra classe.

```
init: function(game){
        this.actions = [];
        this.game = game;

        var thiss = this;

        var his_touchStart= function(evt){
                thiss.onTouchStart(evt);
        };

        var his_touchEnd = function(evt){
                thiss.onTouchEnd(evt);
        };


        var his_mouseDown= function(evt){
                thiss.onMouseDown(evt);
        };

        var his_mouseUp = function(evt){
                thiss.onMouseUp(evt);
        };

        var his_keyDown = function(evt){
                thiss.onKeyDown(evt);
        };

        var canvas = document.getElementById('canvas');

        canvas.addEventListener('touchstart',his_touchStart,false);
        canvas.addEventListener('touchend',his_touchEnd,false);
        canvas.addEventListener('mousedown',his_mouseDown,false);
        canvas.addEventListener('mouseup',his_mouseUp,false);



},
```

Els mètodes més interessants són els de captura del moviment d'un bloc, aquests moviments es produeixen per dos events: touchStart i touchEnd:

```
onTouchStart: function(evt){
        var touch = evt.changedTouches[0];
        this.x = touch.pageX;
        this.y = touch.pageY;
},

onTouchEnd: function(evt){
        var touch = evt.changedTouches[0];

        this.pushAction({
                        action: IOManager.SPLIT,
                                sx: this.x,
                                sy: this.y,
                                dx: touch.pageX,
                                dy: touch.pageY
        });

        if (this.game.getStateId() == Game.START_MENU){
                this.pushAction({action: IOManager.START});
        }
},
```

Quan es produeix un event de touchEnd, necessàriament s'ha produït un event de touchStart que ha capturat les coordenades inicials, només hem de calcular les coordenades finals i crear l'acció "Split" amb totes aquestes coordenades per ser analitzades en un futur.

Mostrem aquí el mètode que analitza aquestes dades, encara que es troba dins de Matrix.

```
actionCellsInPixels: function(sx,sy,dx,dy){
        //console.log(sx,sy,dx,dy);

        var rowA = Math.floor((Math.abs(this.y) + sy + (this.top*64))/64);
        var rowB = Math.floor((Math.abs(this.y) + dy + (this.top*64))/64);

        var colA = Math.floor((sx-this.x)/64);
        var colB = Math.floor((dx-this.x)/64);

        //Mirem que els files i columnes estiguin dins els limits de la matriu de joc
        if (rowA < this.top || rowA > this.floor-1 ||
            rowB < this.top || rowB > this.floor-1 ||
            colA < 0 || colA > this.cols-1 ||
           colB < 0 || colB > this.cols-1) {

        }else{
                if (rowA == rowB){
                        this.actionCells(rowA,colA,rowB,colB);
                }
        }
},
```

Aquest càlcul requereix conèixer la posició relativa de la matriu respecte el canvas i recordem que aquesta es va movent en zig-zag. També requereix conèixer la mida dels blocs i validar que el moviment es fa entre blocs vàlids de forma horitzontal. Finalment, es crida el mètode actionCells qui és qui fa realment l'intercanvi de blocs.

MatrixRenderer

Encarregada de renderitzar la matriu, originalment utilitzava KineticJS, però actualment ho fem de forma manual. Tinc pendent de fer proves de pre-renderització dins un objecte canvas dinàmic, però això ho farem a la fase final.

```
render: function(gameState,debug){

        this.debug = debug;

        if (gameState == Game.IN_GAME){
                for(id in this.playersMatrix){
                        var mObj = this.playersMatrix[id];
                        this.matrix = mObj.getMatrix();
                        this.renderMatrix(mObj);

                this.ctx.globalAlpha = 0.6;
                this.ctx.fillStyle='black';
                this.ctx.fillRect(mObj.getX(),mObj.getY()+(15*64),6*64,
                                        64-(318+mObj.getY()))

                }

                if (this.debug){
                        this.renderDebugGrid(64);
                }
        }
```

```
},
```

Recorrem tots els jugadors i obtenim la seva matriu, finalment cridem renderMatrix. Per cada matriu dibuixem un rectangle ombrejat per representar els blocs nous que pugen i que no estan actius(fillRect). Amb la propietat globalAlpha fem que aquest rectangle sigui translúcid.

```
renderMatrix: function(matrixObj){
        var matrix = matrixObj.getMatrix();
        for (i=0; i< matrix.length; i++){
                for(j=0; j< matrix[i].length; j++){
                        this.renderCell(matrix[i][j],i,j,matrixObj.getX(),
                                matrixObj.getY(), matrixObj.getFloor());
                }
        }
},
```

```
renderCell: function(cell,i,j,iniX,iniY, floor){
        var size = cell.getSize();
        var sx = 0 + cell.getType() * 64;
        var sy = 0;
        var sw = 64;
        var sh = 64;
        var dy = (i *64) + iniY;
        var dx = (j *64 ) + iniX;
        var dw = 64 * size.cols;
        var dh = 64 * size.rows;

        var opacity = 1;

        if (i-1 + size.rows == floor)  dh -=(318+iniY);

        if (this.debug) {
                this.ctx.font = 'italic 10pt Calibri';
                this.ctx.fillText(cell.getType()+"["+this.matrix[i].length+"]",
                                dx+500,dy+30);
                this.ctx.fillText("R:"+cell.isRoot()+":F"+cell.isFixed(),dx+500,dy+60);
                this.ctx.strokeRect(dx+500,dy,62,62);
        }

        if (cell.getType() != 0 && cell.isRoot()){
                this.ctx.globalAlpha = opacity;
                this.ctx.drawImage(this.img,sx,sy,sw,sh,dx,dy,dw,dh);
        }
},
```

Per renderitzar, utilitzem la tècnica que hem explicat anteriorment en la secció “Renderitzat” dins el plantejament del joc.

PlayerRenderer

Renderitza les dades del jugador (marcador) i els efectes associats.

```
render: function(gamestate) {

        if (gamestate == Game.IN_GAME) {
                this.ctx.fillStyle = 'white';
                this.ctx.globalAlpha = 0.7;
                this.ctx.fillRect(0,0,500,30);

                this.ctx.fillStyle = 'black';
                this.ctx.globalAlpha = 1;
                this.ctx.font = '20pt Londrina Solid';
                this.ctx.fillText("Score: "+ this.player.getScore(), 10, 20);
                this.ctx.fillText("Level: "+ this.player.getLevel(), 400, 20);


                this.renderEffects();
        }
},

renderEffects: function(){
        var effects = this.player.getEffects();

        for(i = 0; i< effects.length; i++){

                effects[i].render(this.ctx);
        }

}
```

Veiem que la lògica de renderització la definirà cada efecte per sí mateix, donat que poden ser molt diferents segons el tipus d'efecte que es vol representar, mostrem l'exemple del render que funcionaria pel efectes de tipus "addScore".

```
render: function(ctx){
        if (this.name == 'addScore'){
                ctx.font = 'italic 20pt Calibri';
                ctx.lineWidth   = 1;
        }else{
                ctx.font = 'italic 50pt Calibri';
        }

        ctx.fillText(this.value, this.dx,this.dy);

        ctx.strokeStyle = 'white';
        ctx.lineWidth   = 2;

        ctx.strokeText(this.value, this.dx,this.dy);
},
```

dx i dy són dos coordenades que s'aniran actualitzant automàticament el que crearà l'efecte de pujar cap al cel.

Desplegament (Deploy)

El desplegament consisteix en empaquetar el codi font en un format optimitzat per ser executat de forma nativa dins els dispositius mòbils, és a dir, com una aplicació instal·lada al dispositiu sense utilitzar navegador web.

Aquesta tasca no ha resultat gens senzilla, doncs m'he trobat amb molts problemes de rendiment executant el joc en diferents mòbils de gama mitjana.

Així doncs, veurem que la solució que la tecnologia PhoneGap que vam seleccionar inicialment no ens donarà els resultats esperats i haurem de provar altres alternatives.

PhoneGap: Primera ensopegada.

PhoneGap és una solució open-source multiplataforma que ens permet desplegar aplicacions fetes amb HTML, Javascript i CSS sobre diferents plataformes mòbils: IPhone, Android, Blackberry, WebOs, Windows Phone, Symbian i Bada. PhoneGap en el seu nucli utilitza Apache Córdoba per accedir de forma nativa als diferents recursos: acceleròmetre, càmera, etc.

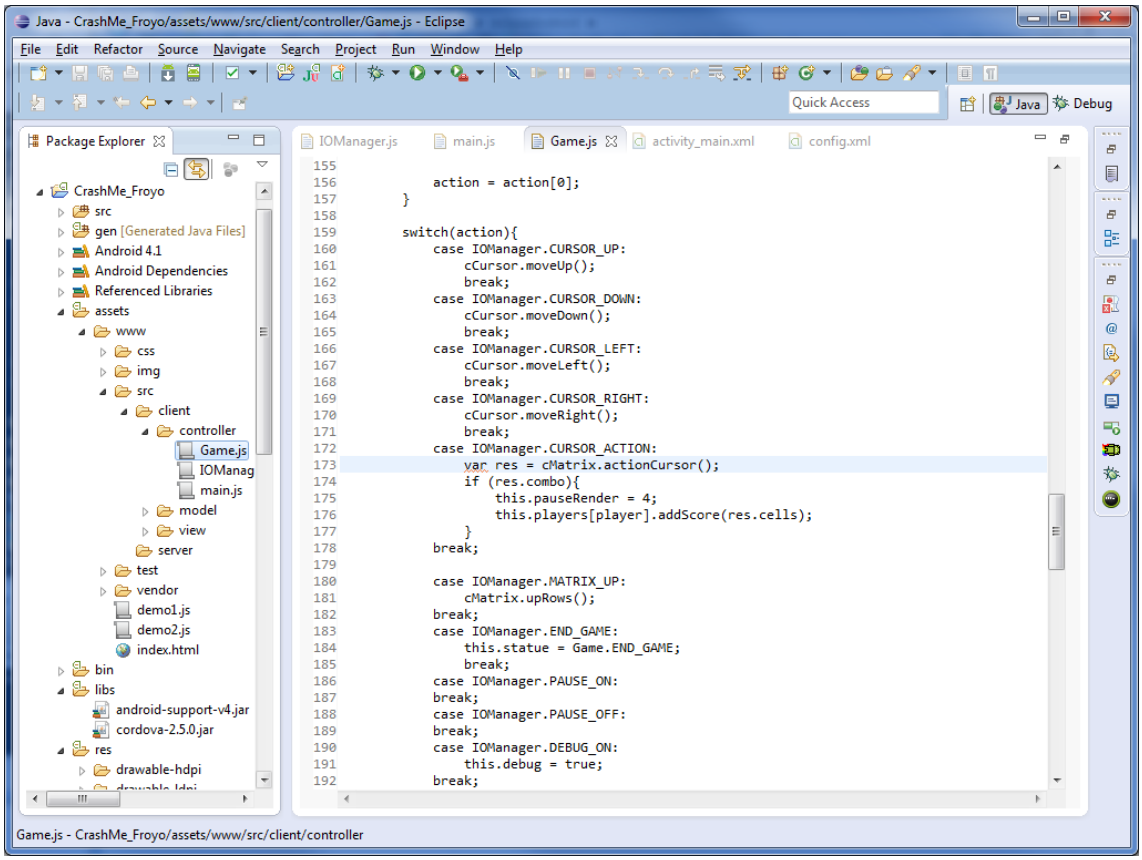
En el meu cas, he fet les proves de desplegament sobre Android i la forma més senzilla d'utilitzar-ho és com a plugin d'Eclipse juntament amb els plugins del ADK (Android Development Toolkit).

El que s'ha de fer és crear un projecte mòbil, afegir els nostres fitxers dins un directori determinat, modificar alguns fitxers de configuració segons unes pautes establertes i afegir la llibreria d'apache cordoba al projecte.

No inclouré en aquesta memòria un tutorial de com fer aquest desplegament, a la bibliografia he inclòs un parell de referències en les que he basat la meva solució.

Com podem veure, encara que no és un procés automatitzat, no és gaire complex, es tracta d'anar amb cura modificant diferents fitxers i podrem executar-la com si d'una aplicació nativa es tractés. La sorpresa ens la trobem al final: **el joc va molt lent**!.

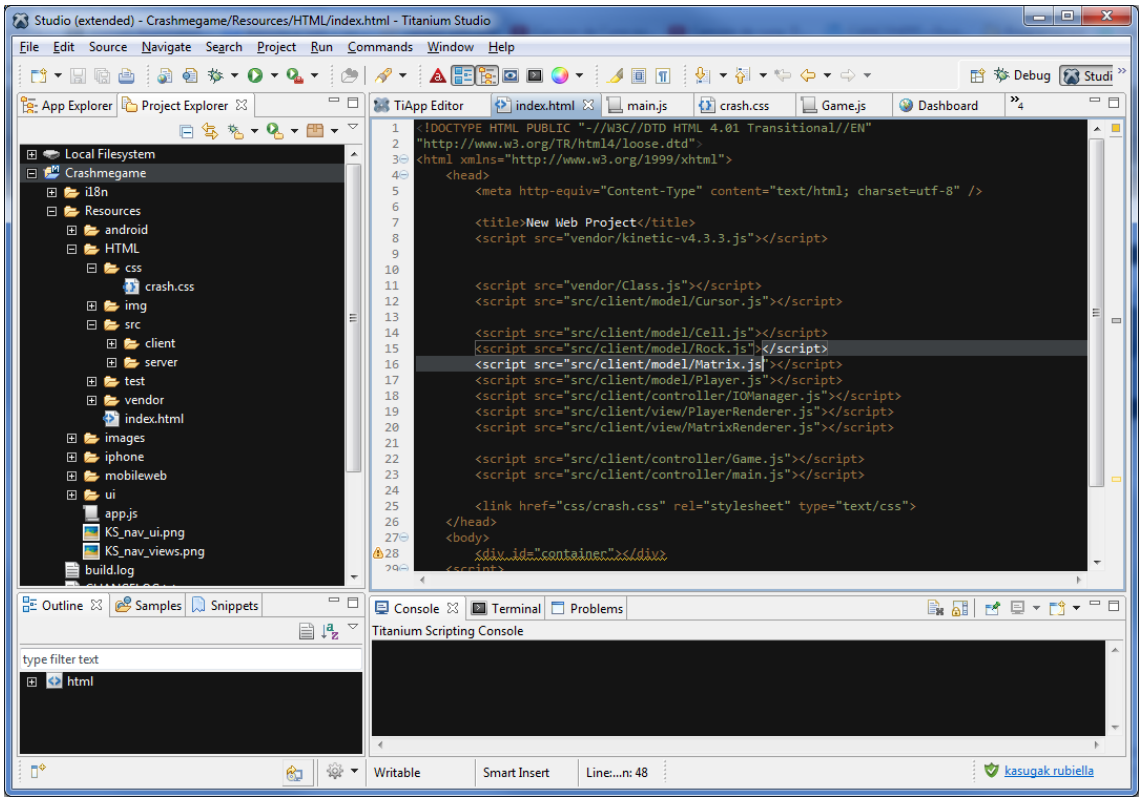
El nostre joc, encara està en fase de desenvolupament, és a dir, segurament no està gaire optimitzat, però tampoc hem inclòs animacions, sons, multijugador, ..., així que considerem que o bé tenim un problema greu de programació o aquesta solució no ens serveix.



Titanium : Segona ensopegada.

Titanium SDK és una solució similar a PhoneGab, però més optimitzada. En aquest cas, és molt més fàcil d'utilitzar que PhoneGap doncs només ens hem de descarregar el seu IDE basat en Eclipse: Titanium Studio, on tenim tot configurat i preparat per executar la nostra aplicació.

El desplegament consisteix en copiar tot el contingut del nostre codi dins un directori específic (HTML) i seleccionar el dispositiu destí.



Encara que no he disposat d'una eina per mesurar el rendiment per comparar-lo amb PhoneGap, si que sembla que el joc funciona una mica més fi, però continua sense poder-se jugar.

Arribats a aquest punt, no tenim altre remei que analitzar què passa amb el nostre joc i per què no funciona com deuria, no té sentit continuar desenvolupant, si després no funcionarà correctament.

Investigació Rendiment

En buscar per internet veiem que no som els únics amb aquests problemes, sembla ser que aquestes eines funcionen molt bé per aplicacions estàndard, però no donen bons resultats amb jocs. El principal problema el tenim al manipular l'objecte canvas, que requereix de recursos específics pel renderitzat i, com ja hem vist, el jocs requereixen renderitzar la pantalla contínuament (FPS), una aplicació normal no té aquestes necessitats específiques i pot funcionar prou bé amb les tecnologies que hem provat (PhoneGap i Titanium).

Al blog de scirra.com han fet una anàlisi força exhaustiva. L'anàlisi consisteix en veure quina és la velocitat de renderitzat per segon (FTP) del mateix joc desplegat en diferents dispositius amb diferents tecnologies, la url complerta a l'anàlisi, la tenim a la bibliografia, però mostrarem aquí la taula resultant:

Dispositiu	Tecnologia	FPS
Desktop	Chrome 19	60
Desktop	Firefox 13	64
Desktop	IE9	60
Desktop	Safari 5.1	26
Desktop	Opera 12 beta	27
iPhone 4S	Safari	34
iPhone 4S	**PhoneGap**	27
iPhone 4S	directCanvas	52
iPad 2	Safari	42
iPad 2	PhoneGap	14
iPad 2	directCanvas	53
iPad 3	Safari	30
iPad 3	**PhoneGap**	3
iPad 3	directCanvas	54
Android 4	Stock browser	11
Android 4	PhoneGap	10
Android 4	Chrome 18 beta	24
Android 4	Firefox 14 beta	7
Android 4	Opera 12 mobile	10
Android 4	**CocoonJS**	34
Windows Phone 7	IE Mobile	14
Windows Phone 7	**PhoneGap**	10
PlayBook	Browser	9
PlayBook	Browser	30

Considerem que el joc necessita uns 30 FPS per poder jugar sense problemes (color verd) i, per sota d'aquest límit, podem notar sotragades o dificultats.

Com veiem PhoneGap no arriba mai als 30 FPS, la seva millor puntuació és 24 FPS en un dispositiu de gama alta com és l'iPhone 4S, però també uns lamentables 3 FPS en una tablet potent com és iPad 3.

Ens crida l'atenció una nova tecnologia: CocoonJS, encara que l'anàlisi no ho contempla, també funciona sobre IOS i sembla donar uns resultats acceptables sobre Android.

Abans però, de tornar a ensopegar-nos, analitzarem amb més cura si podem millorar el nostre codi perquè sigui més eficient.

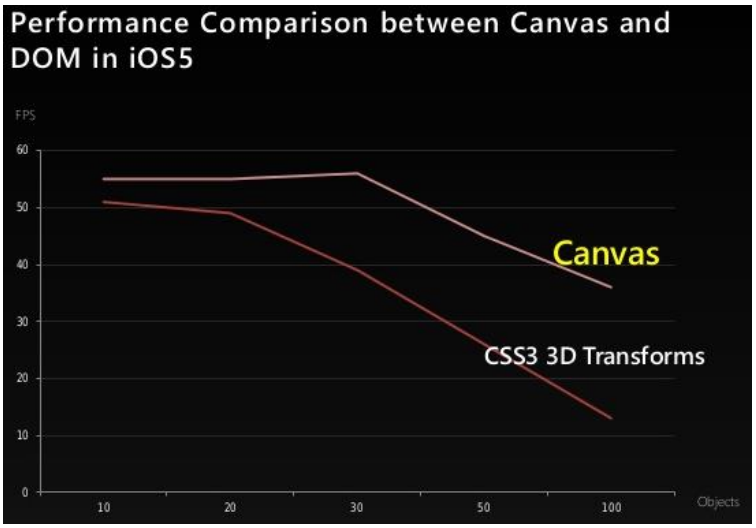
Optimitzant Canvas

Ja hem parlat de per què l'objecte canvas és el més problemàtic per portar-ho als mòbils i ens adonem que aquí podem millorar el nostre codi. Fins ara, havia confiat la renderització a la biblioteca KineticJS i això ens implica certes limitacions:

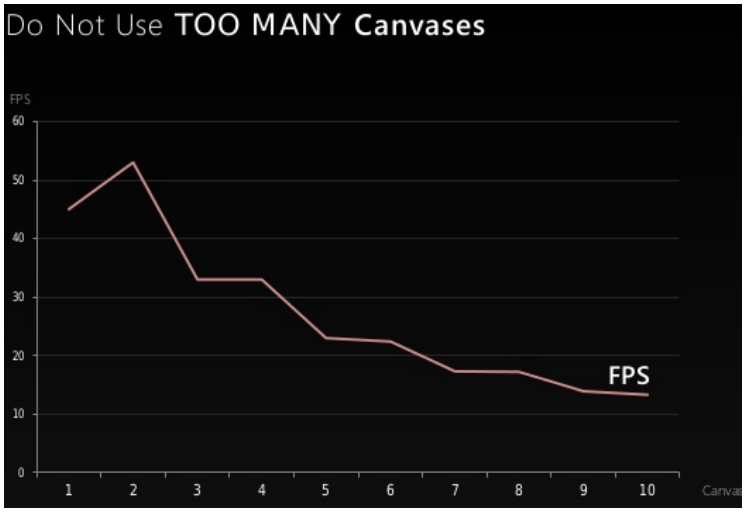
- Les capes són implementades per objectes canvas independents: més capes, més canvas.
- No tenim control exhaustiu sobre les imatges ni sobre la renderització.

A slideshare hi ha una presentació força interessant "High Performance Mobile Web Game Development in HTML 5", on podem extreure dos punts molt interessants, veiem aquestes gràfiques:

Comparació entre Canvas i CSS3+DOM i FPS.



Comparació entre número de canvas i FPS.



A la primera gràfica se'ns mostra que canvas és més eficient que utilitzar CSS per manipular el DOM de la pàgina. Inicialment, havia plantejat dibuixar alguns elements amb CSS per evitar l'ús de canvas, però la gràfica ens mostra clarament que és un error.

La segona gràfica ens mostra que si tenim diferents objectes canvas, el rendiment cau en picat, per tant, millor no utilitzar les capes de KineticJs, ja que cada capa s'implementa com un canvas independent.

En altres transparències també se'ns mostra la ineficiència de manipular objectes i rotar-los dins un canvas, sent molt més eficient disposar de sprites que simulin aquests desplaçaments.

Així doncs, he decidit prescindir de la biblioteca KineticJS i reimplementar el renderitzat de forma totalment manual. Un altra característica que podrem implementar al fer-ho a mà és la pre-renderització en un objecte dinàmic.

Gràcies a que teníem la classe de renderització separada de la resta (MatrixRenderer) aquesta feina ha estat prou senzilla, a la part d'anàlisi de codi hem vist més exhaustivament com s'ha implementat.

El nostre salvador: CocoonJS

CocoonJS és un framework especialitzat en videojocs completament diferent a la resta de tecnologies que havíem provat fins ara. El seu motor es centra en optimitzar al màxim l'objecte canvas utilitzant l'acceleració OpenGL nativa de cada dispositiu.

No és compatible amb CSS i no és 100% compatible amb tots els events Javascript ni tots objectes HTML, però si ho és amb els principals events i objectes que intervenen en el desenvolupament d'un videojoc: canvas, events de touching, sons i fonts.

Per altra banda, inclou un sistema automatitzat per publicar a l'Apple Store o Google Play i de monetitzar el nostre joc amb publicitat.

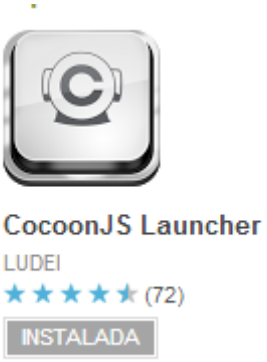
Aquestes limitacions m'han obligat a fer alguns canvis en el codi Javascript, no gaire importants i a prescindir completament del CSS.

Desplegament amb CocoonJS

Hi ha dues formes de fer el desplegament, si volem generar una aplicació compatible amb les "Stores" o si simplement volem debugar l'aplicació al nostre mòbil.

Mode Depuració

Per depurar (debug) ens hem d'instal·lar una aplicació específica al nostre dispositiu, aquesta aplicació la tenim disponible en les Stores de Google i Apple. Aquesta aplicació s'utilitzarà com a llançadora o "Launcher".



Una vegada instal·lat el Launcher haurem d'enregistrar-nos de forma gratuïta i obtindrem una clau única com a desenvolupadors.

Ara l'únic que haurem de fer és comprimir el nostre codi html + Javascript dins un fitzex zip i copiar-ho a l'arrel de la nostra memòria SD del telèfon mòbil.

El Launcher ens mostra un llistat de tots el jocs que pot executar dins la nostra memòria i podem seleccionar-lo i executar-ho.

Com podem veure amb més detall al vídeo de la demostració, disposa d'una consola de debug compatible amb Javascript (console.log) que ens serà molt útil.

Depurant amb Android

Android disposa de forma nativa d'un mode de debug molt complert. Podem activar diferents opcions que ens permet visualitzar en temps real la detecció d'events sobre la pantalla (touching), consum de memòria, estat del processador, pila d'execució, coordenades de la pantalla, etc.

Combinant això amb la consola de CocoonJS disposem d'un entorn de prova molt potent sobre un entorn "real", ja que ho podem fer sobre qualsevol dispositiu android.

Reescalat Automàtic

CocoonJs té encara una avantatge addicional i és que tot sol ens re-escalarà a pantalla completa el joc i ens calcularà les coordenades correctament, això que pot semblar molt senzill, en el fons és una gran avantatge ja que gestionar manualment la gran diversitat de resolucions, mides de pantalla, ... que hi ha actualment al mercat, pot resultat una autèntica bogeria.

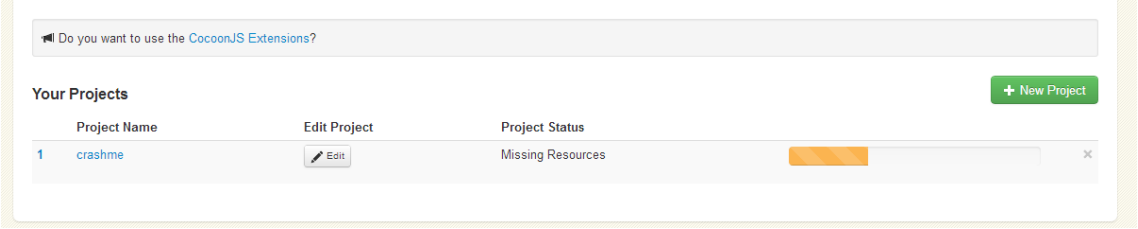
Juntament amb la presentació de la memòria inclouré un vídeo demostració del desplegament del nostre joc en mode debug amb CocoonJS.

Desplegament a Google Play o Apple Store.

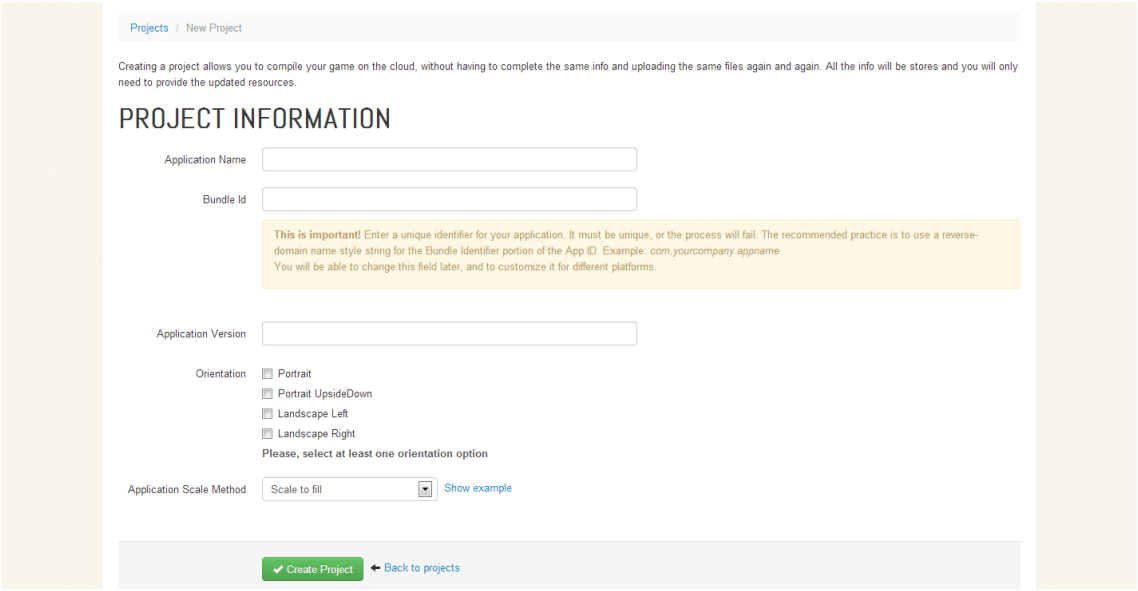
Per poder oferir el nostre joc a una d'aquestes plataformes primer és necessari processar el codi (compilació), per tal que sigui executable sense requerir de cap "Launcher" especial, com fèiem quan estàvem debugant.

Per dur a terme aquesta feina CocoonJS ens ofereix de forma gratuïta l'ús d'una plataforma cloud (Amazon cloud) per poder gestionar, compilar i publicar els nostres projectes de forma fàcil i senzilla a través d'una eina web.

A la pàgina principal tenim el llistat dels nostres projectes i podem crear-ne de nous:



Crear un nou projecte és molt senzill:

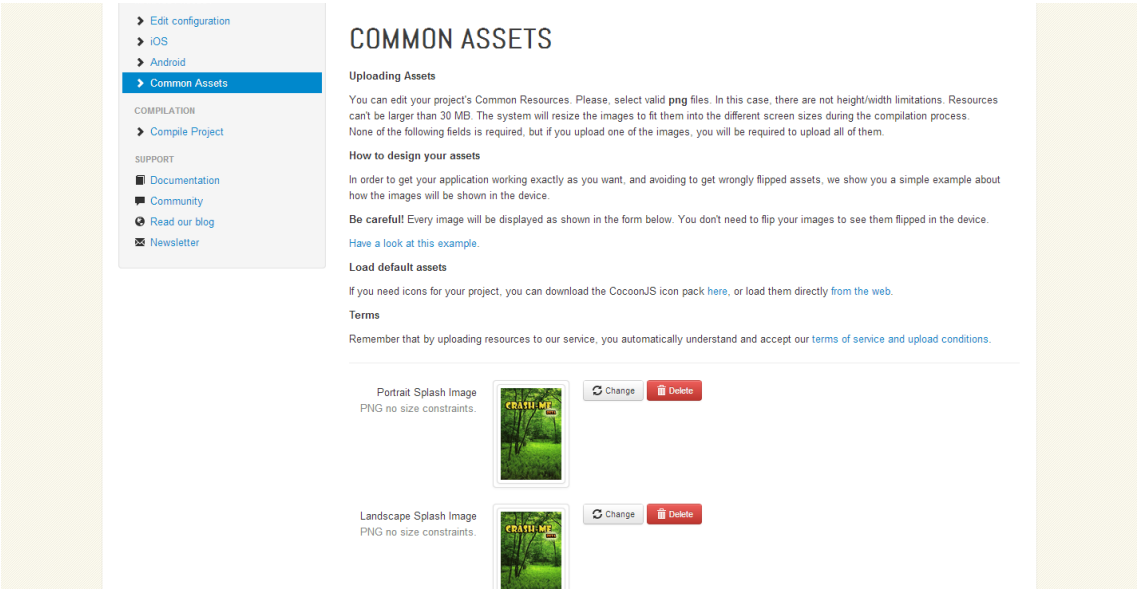


Segons les opcions que seleccionem, ens gestionarà el reescalat automàtic per omplenar tota la pantalla o per omplenar mantenint les proporcions.

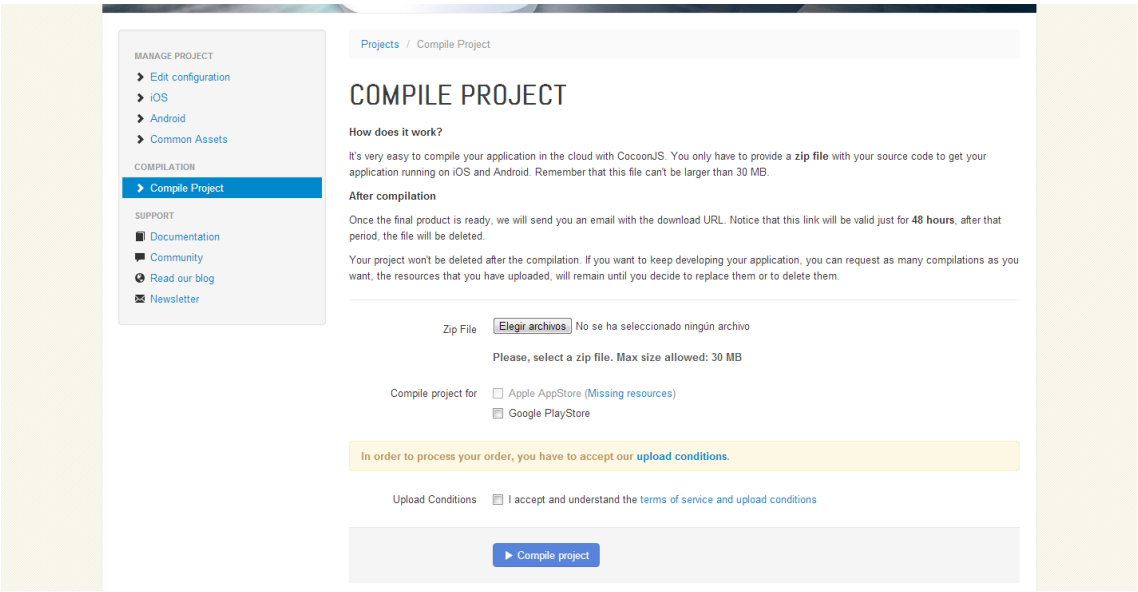
També ens demana que seleccionem l'orientació de la pantalla de mode que ens fixarà una orientació i així no ens haurem de preocupar-nos del canvi d'orientació automàtica que utilitzen la majoria de dispositius.

Finalment, al panell de configuració del projecte haurem de pujar les icones i pantalles d'inici (splash) que vulguem utilitzar, a tot això se li anomena "assets".

El format i la mida dels assets pot ser depenent de la plataforma destinatària (Android o IOS), així que disposem d'opcions separades on anar pujant les nostres imatges:



Finalment, disposem d'una pantalla on pujar el nostre codi i compilar-ho:



El zip que hem de pujar és el mateix zip que utilitzem per debugar. Una vegada pujat el codi i seleccionada la plataforma destí (AppStore o PlayStore), haurem d'esperar a que la compilació finalitzi.

Quan el cloud hagi compilat el nostre codi disposarem d'una URL amb la que podrem descarregar el joc preparat per ser executat en qualsevol dispositiu.

Simulació Costos

En aquest apartat farem una simulació dels costos reals que podria tenir un projecte d'aquestes característiques. Per aquesta simulació definirem una sèrie de rols i costos segons el rol desenvolupat.

Treballarem sobre dos tipus de costos: l'equipament i l'humà i suposarem que la durada total del projecte és de un mes.

Respecte el cost humà, aquest depèn dels rols i la dedicació de cada un d'aquest rols, per tant definirem primer els rols:

Rol	Funcions	Cost
Cap de projecte	Planificació Gestió de recursos Scrum Master Validador	60 €/h
Analista	Definició del requeriments Anàlisis funcional Diseny solució software	40 €/h
Disenyador	Diseny interfícies Usabilitat Animacions	30 €/h
Programador	Desenvolupament Tests Documentació	20 €/h

Cost humà segons dedicació

Rol	**Dedicació (h)**	**Cost (€)**
Cap de projecte	10	600
Analista	50	200
Disenyador	80	2400
Programador	180	3600
Total		6800

Cost d'equipament:

Equipament	Definició	Cost
1x Oficina	Zona de treball	300€/mes
2x ordinadors portàtils	Gama mitja.	1500 €
2x ordinadors sobretaula	Gama alta.	2000 €
Material de Oficina	Llibretes, post-its, bolígrafs, paper, etc..	50 €
Pissarra gran magnètica	Per tenir un scrumban fer diagrames, etc.	100€
Mobiliari	Mobiliari bàsic per treballar	200 €
Electricitat	Consum electric	100 €/mes
Llicencies Software	Software lliure: Ubuntu, Eclipse, etc..	0
Total		4250

Si volem fer una estimació una mica més realista hauríem d'associar només el cost de "desgast" de certs materials, per exemple:

Suposem que els ordinadors tenen una vida útil de 5 anys amb un manteniment anual de 50 euros i només els utilitzem durant un mes:

Cost mensual ordinadors = (3500 + (30 * 5)) / (5*12) = 60€

Suposem que el mobiliari i pissarra magnètica tenen una vida útil de 20 anys:

Cost mensual Mobiliari + pissarra = 300/(20*12) = 1.25 €

Revisem el cost Final

Recuros	Cost Final (€)
Cost Humà	6800
Cost Desgat i lloguer equipament (1 MES)	511
Total	**7311**

Finalment, podem concloure que el cost final d'aquest projecte seria d'uns **7300** euros.

Conclusions finals

Ha sigut una experiència molt enriquidora on he après moltes coses noves. La programació d'un videojoc és força diferent a una aplicació d'escriptori o un aplicació web tradicional i, a més, he pogut aprofundir en el llenguatge Javascript i el seu ús amb pseudo-objectes.

A mig projecte em vaig trobar en una situació força complexa quan veia que el videojoc no funcionava correctament als telèfons mòbils. Això m'ha obligat a invertir més hores de proves i investigació de les esperades per fer el desplegament. Entendre el problema i trobar-hi una solució, ha sigut molt satisfactori.

Respecte el videojoc en sí mateix, crec que s'ha donat amb una solució força genèrica i modular, seguint patrons de disseny i bones pràctiques de programació amb un codi força net i senzill el que permetria un fàcil creixement i extensió.

Javascript s'està imposant com estàndard cada vegada amb més força, i ja no només a nivell de client sinó a nivell de servidors amb l'exitós NodeJS. Crec que seleccionar aquest projecte ha sigut una decisió molt encertada que de ben segur que aquesta experiència m'ajudarà en el món laboral.

Bibliografia

Llibres	Autor
HTML5 Mobile Development Cookbook	Shi Chuan
Game Engine Architecture 2012	Jason Gregory
Pro HTML5 Games	Aditya Ravi Shankar
Professional HTML5 Mobile Game Development	Pascal Retting
Game Coding Complete	Mike McShaffry

Pàgines Web
Comparativa entre PhoneGap i Titanium: http://www.universalmind.com/blog/mobile-html5-phonegap-vs-appcelerator-titanium
Anàlisis d'eficiencia de jocs sobre diferents plataformes: https://www.scirra.com/blog/85/the-great-html5-mobile-gaming-performance-comparison
Blog on ens mostra com utiltizar CooconJS: http://www.nazariglez.com/2013/02/26/exporta-tus-juegos-html5-a-plataformas-moviles-con-cocoonjs/
Game Loop amb Javascript: http://nokarma.org/2011/02/02/javascript-game-development-the-game-loop/index.html
Anàlisis rendiment canvas: http://www.slideshare.net/iiwork/high-performance-mobile-web-game-development-in-html5
Objectes i herència amb Javascript: http://ejohn.org/blog/simple-javascript-inheritance/
Tutorials i Manuals HTML5 i Javascript: http://www.html5rocks.com http://www.html5rocks.com/en/tutorials/canvas/performance/
Curs Gratuit d'iniciació a la Programació de Videojocs: https://www.udacity.com/course/cs255

Software	URL
Aptana IDE	http://www.aptana.com/
Titanium IDE	http://www.appcelerator.com/
Chrome Browser	https://www.google.com/intl/es/chrome/browser/

Subversion (CVS)	http://subversion.tigris.org/

Frameworks/APIS	URL
PhoneGap	http://phonegap.com/
CocoonJS	http://www.ludei.com/tech/cocoonjs
ImpactJS	http://impactjs.com/
Collie	http://jindo.dev.naver.com/collie/
KineticJS	http://kineticjs.com/
GameClosure	http://www.gameclosure.com/
CreateJS	http://www.createjs.com

Glossari

FPS: Frames Per Second, es refereix al número d'actualitzacions de pantalla que es fan per segon.

Bucle: Porció de codi que es repeteix de forma indefinida o sota una condició de fi.

Iteració: Una execució completa d'un bucle.

Desplegament: Procés de instal·lar l'aplicació dins al client destí.

Framework: Conjunt d'eines o API's.

Canvas: Etiqueta especial d'HTML5 que ens permet renderitzar objectes amb Javascript.

IDE: Entorn de Desenvolupament que integra diferents eines.

Debug: Mode de proves i depuració.