Code Invaders

Proiect realizat de Manaila Andrei Petrut și Fovas Denis Daniel George

Cuprins

Code invaders

Începutul proiectului

Cu toții avem avut parte de mici jocuri în copilărie, iar pentru noi, Fovas Denis și Manaila Andrei, jocul care ne-a marcat cel mai mult a fost „Chicken Invaders”. Cu gândul la copilărie, am început munca la acest joc.

Interfata jocului

Tehnologia folosită pentru creeare acestuia a fost bazata pe HTML 5, iar mai exact pe tehnologia de tip „canvas”. Canvas-ul este asemănător unei pânze de pictura, de unde i se trage și numele, astfel ca putem folosi anumite tehnici într-un mod mai ușor și mai lejer pentru a creea partea grafica. Totodata folosim și Jquery pentru anumite funcții destul de esentiale și pentru a preveni anumite probleme posibil ușor de întâlnit pe viitor.

Tema aplicației

Pe lângă dorința de a avea un joc pe baza Chiken Invaders, am dorit să facem jocul și asemănător într-un fel cu o asemanare cu programare. Mai exact, spre deosebire de modul clasic în care, inamicii sunt dușmanii principali, jucătorul trebuie sa ii distrugă, noi am dorit o abordare puțin mai diferita. „Inamicii” sunt reprezentați de limbaje de programare iar jucătorul este un programator care are misiunea de a crea un program super eficient și perfect folosind doar: C++, C#, php, Java, iar la final are de terminat informațiile principale ale jocului, catalogate drept: Big Data.

Ca și orice program, apar multe bug-uri pe durata dezvoltarii acestuia, iar programatorul are datoria de a distruge aceste bug-uri pana sa afle șeful. Dacă cumva se găsesc 100 de bug-uri, șeful afla iar programatorul este concediat. Programatorul nu vrea sa își piarda locul de munca, dar și este nevoit de a-si termina treaba.

Programul este stabilit în 4 nivele care va avea nevoie de mai multe limbaje de programare. Fiecare limbaj de programare are nevoie de anumite secvente de cod care pot fi fie bune, fie rele. Limbajele de programare o să arunce unul din cele 3 tipuri de linii de cod, dintre care 2 tipuri sunt linii rele de cod, linii care o să provoace bug-uri iar doar una este linie buna de cod, care poate rezolva bug-uri. Utilizatorul trebuie sa termine sarcinile adresate asupra limbajelor de programare dezvoltate, dar totodata trebuie sa și se fereasca de eventualele bug-uri. Dacă intampina ceva bug-uri, atunci are șansa de a găsi rezolvarea în cursul dezvoltarii aplicației, lucru reprezentat prin liniile de cod bune. Dacă cumva apar prea puține bug-uri, atunci nu se v-or observa și utilizatorul va scapa nepedepsit. Dacă liniile de cod sunt linii rele, atunci ele o să adauge un anumit numar de bug-uri la program, iar numărul de bug-uri o să depinda de la nivel la nivel, în funcție de task.

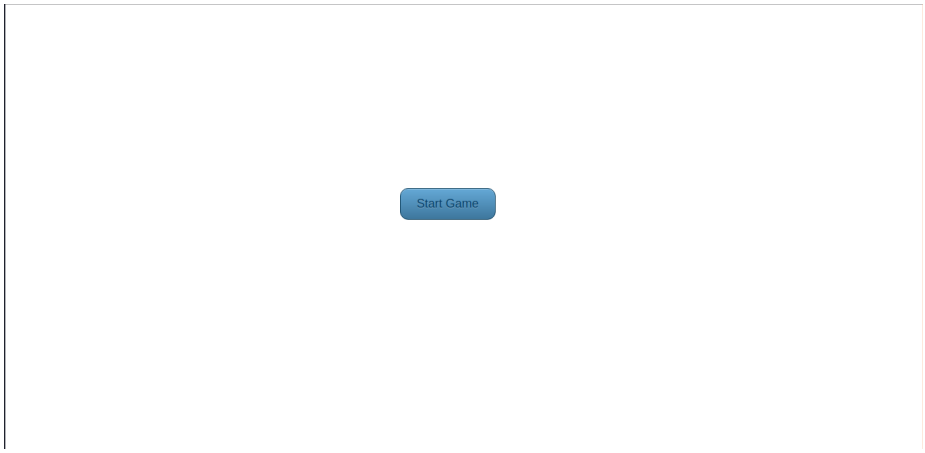


Figura 1.

Aspectul paginii index.html

```
<html >
<head>
    <!-- Proprietati CSS proprii -->
    <link rel="stylesheet" type="text/css" href="Style/style.css">
    <title>Code invaders</title>
</head>
<body>
    <a href='mainGame.html' class='button centrat'>Start Game</a>

</body>
</html>
```

Figura 2.

Codul paginii index.html

```
.centrat {
    position: absolute;
    margin-top: 20%;
    left: 43%;
}

canvas {
    position: absolute;
}

.game-centrat {
    position: absolute;
    margin-top: 2.4%;
    left: 19%;
}
```

Figura 4.

Cod CSS pentru a centra butonul, cât şi pentru a putea centra canvas-urile

Gameplay

În cadrul jocului, utilizatorul este reprezentat de nava sa, o nava de tip 8-bit, care trebuie sa avanseze în cadrul jocului. Singurul mod de a avansa este reprezentat prin termianrea secventei de cod, lucru reprezentat prin „distrugerea inamicului”. În momentul în care secvența de cod este terminata, ea dispare de pe ecran. Pentru a putea termina secvența de cod, utilizatorul va trage un proiectil. În momentul în care se nimereste o secvența de cod, este eliminata o parte din cerinta sa, lucru reprezentat prin scaderea barii de viața afisate dedesuptul acesteia. „Inamicii” o să returneze la compilare un cod, care paote fi fie bun, fie rău. Aceasta compilare este reprezentata prin modul în care inamicii trag liniile de cod. În momentul în care utilizatorul a terminat toate task-urile, o să avanseze la următorul nivel. Primele 3 nivele reprezintă task-uri mici și multe. Aceste task-uri au nevoie de mai multe incercari pentru a fi terminate:

- C++ are nevoie de o încercare
- C# are nevoie de 2 incercari
- php are nevoie de 3 incercari
- Java are nevoie de 5 incercari

După aceste 3 nivele, urmează ca utilizatorul sa termine task-ul final, denumit drept „Big Data” reprezintă modul în care utilizatorul introduce toate datele necesare în program, dar necesita mult efort, pentru a putea fi introduse corect. Astfel ca, utilizatorul are nevoie de multe incercari pentru a putea termina acest task.

Primul nivel pune inamicii într-un mod static, într-un fel în care inamicii nu se mișca, iar numărul de bug-uri oferit de inamici este de 10 bug-uri/linie de cod rea. Al doilea nivel ofera un numar mai mare de inamici și o mișcare de sus-jos. O alta trasatura a acestui nivel este aceea de a pune inamicii într-un mod relativ aleator. Nivelul 3 păstrează trasatura inamicilor de a se afla într-un mod aleator, dar o să ofere și o mișcare relativ aleatorie a lor, astfel ca ei se mica fie sus-jos, fie stânga dreapta, mișcare care o păstrează pana în momentul în care sunt distruși. În cadrul acestui nivel, nivelul de bug-uri este crescut la 30 de bug-uri. Numărul de nave din cadrul nivelului 4 este de doar un inamic, inamic care reprezintă *boss*-ul nivelului. Acesta are 500 de lovituri ca să fie terminat, și lansează un numar mare de linii de cod rele. Liniile de cod apar în mod aleatoriu sub acest inamic, și nu ofera linii de cod care rezolva bug-uri. Odată ajuns la acest nivel, numărul de bug-uri va crește doar, nefiind o șansa de a elimina numărul de bug-uri. După terminarea acestui nivel, jocul se termina, utilizatorul find în cadrul nivelului 5. În acest moment este prezentat pentru utilizator mesajul *„Ai câștigat!”*. În caz contrar, în caz ca utilizatorul acumuleaza 100 de bug-uri, sau chiar mai multe, atunci o să i afiseze mesajul *„Ai pierdut!...”*. Pentru oricare dintre aceste rezultate, după expirarea unui timp de 5 secunde, pagina se va reincarca, iar jocul va reincepe de la început.

Interfata

Acest joc începe cu o pagina goala, iar în centru, un buton „Start Game”. Acesta reprezintă un link către pagina denumita „mainGame.html”, pagina în care o să fie introdus jocul, tot relativ în centru. Dimensiunile jocului sunt de 800x600 pixeli, astfel ca afisarea acestuia nu va fi o problemă pentru mare parte din ecranele disponibile în momentul de fata. Jocul este format pe baza mai

Figura 3.

```css
.button {
   border: 1px solid #0a3c59;
   background: #3e779d;
   background: -webkit-gradient(linear, left top, left bottom, from(#65a9d7), to(#3e779d));
   background: -webkit-linear-gradient(top, #65a9d7, #3e779d);
   background: -moz-linear-gradient(top, #65a9d7, #3e779d);
   background: -ms-linear-gradient(top, #65a9d7, #3e779d);
   background: -o-linear-gradient(top, #65a9d7, #3e779d);
   background-image: -ms-linear-gradient(top, #65a9d7 0%, #3e779d 100%);
   padding: 12px 24px;
   -webkit-border-radius: 12px;
   -moz-border-radius: 12px;
   border-radius: 12px;
   -webkit-box-shadow: rgba(255,255,255,0.4) 0 1px 0, inset rgba(255,255,255,0.4) 0 1px 0;
   -moz-box-shadow: rgba(255,255,255,0.4) 0 1px 0, inset rgba(255,255,255,0.4) 0 1px 0;
   box-shadow: rgba(255,255,255,0.4) 0 1px 0, inset rgba(255,255,255,0.4) 0 1px 0;
   text-shadow: #7ea4bd 0 1px 0;
   color: #06426c;
   font-size: 18px;
   font-family: helvetica, serif;
   text-decoration: none;
   vertical-align: middle;
}

.button:hover {
   border: 1px solid #0a3c59;
   text-shadow: #1e4158 0 1px 0;
   background: #3e779d;
   background: -webkit-gradient(linear, left top, left bottom, from(#65a9d7), to(#3e779d));
   background: -webkit-linear-gradient(top, #65a9d7, #3e779d);
   background: -moz-linear-gradient(top, #65a9d7, #3e779d);
   background: -ms-linear-gradient(top, #65a9d7, #3e779d);
   background: -o-linear-gradient(top, #65a9d7, #3e779d);
   background-image: -ms-linear-gradient(top, #65a9d7 0%, #3e779d 100%);
   color: #fff;
}

.button:active {
   text-shadow: #1e4158 0 1px 0;
   border: 1px solid #0a3c59;
   background: #65a9d7;
   background: -webkit-gradient(linear, left top, left bottom, from(#3e779d), to(#3e779d));
   background: -webkit-linear-gradient(top, #3e779d, #65a9d7);
   background: -moz-linear-gradient(top, #3e779d, #65a9d7);
   background: -ms-linear-gradient(top, #3e779d, #65a9d7);
   background: -o-linear-gradient(top, #3e779d, #65a9d7);
   background-image: -ms-linear-gradient(top, #3e779d 0%, #65a9d7 100%);
   color: #fff;
}
```

Cod CSS pentru butonul „Start Game”

```html
<!DOCTYPE HTML>
<html>
    <head>
        <meta charset="utf-8"/>
        <title>Code Invaders</title>
        <link rel="stylesheet" type="text/css" href="Style/style.css">
        <script src="Scripts/JQuery/jquery-1.2.6.min.js"></script>
    </head>

    <body>

        <div class="container">
            <div class="game-centrat">

        <!-- CANVAS AREA -->
            <canvas id="background" width="800" height="600"  onkeydown = "keyDown(event)" onkeyup = "keyUp(event)">
              <!-- In caz ca acest mesaj apare pe ecran, trebuie utilizat un browser mai modern. -->
                Din pacate browser-ul dumneavoastra preferat, nu accepta tehnologia de tip "canvas".
                Va rugam sa alegeti un alt browser.
            </canvas>
            <canvas id="bullet" width="800" height="600"></canvas>
            <canvas id="enemyBullet" width='800' height='600'> </canvas>
            <canvas id="inamici" width="800" height="600"></canvas>
            <canvas id="action" width="800" height="600"></canvas>
            <canvas id="text" width="800" height="600"></canvas>
            </div>
        </div>

        <!--  END OF CANVAS AREA -->

        <!-- Aici vor fi script-urile -->
            <script type="text/javascript" src="Scripts/script.js"></script>
        <!-- Sfarsit zona script-uri -->

    </body>
</html>
```

Figura 5.

Cod HTML pentru al paginii mainGame.html

multor canvas-uri suprapuse, care ofera imaginea unui singur ecran, care conține toate elementele afisate pe ecran. Aceste elemente, atât butonul de pornire, cât și canvas-urile, sunt introduse în pagini de tip HTML 5.

Continutul paginii „index.html” este prezentat în figura 1, iar codul acesteia este prezentat în figura 2. Pentru buton, avem un link către pagina mainGame.html, predefinit cu clasele de: buton, centrat. Aceste clase ofera aspectul butonului, și proprietatile astfel încât să fie centrat. Aceste proprietăți sunt retinute în fișierul „Styles/style.css”, fisier de tip CSS în care am păstrat codul pentru design-ul celor 2 pagini. În figura 3 avem codul disponibil pentru buton, iar în figura 4 avem codul pentru a pozitiona toate canvas-urile suprapuse, cât și proprietatile ca „butonul” să fie centrat. Prin proprietatile „.button” sunt definite proprietăți valabile pentru toate tipurile de browsere moderne, iar prin proprietatea *.centrat* butonul este pus într-o poziție relativ centrala. Prin proprietatea *.game-centrat* canvas-urile sunt suprapuse una peste alta, și totodata într-o poziție relativ centrata.

Canvas-urile sunt pozitionate într-o pagina diferita fata de cea în care se afla butonul, astfel ca ele se afla în cadrul paginii *mainGame.html*, pagina care conține legătura cu script-ul scris, Jquery și fișierul css descris anteior. Pentru a afisa toate informațiile necesare, folosim 6 canvas-uri separate, fiecare cu câte un rol:

- background – este folosit pentru a putea forma fundalul jocului
- bullet – este folosit pentru a forma proiectilele utilizatorului
- enemey bullet - este folosit pentru a forma proiectilele inamicilor
- inamici – este folosit pentru a putea forma inamicii pe care jucătorul trebuie sa ii loveasca
- action – canvas dedicat afisarii navei utilizatorului
- text – canvas dedicat pentru a putea afisa scorul și nivelul cât și rezultatul jocului (câștig sau pierdere)

În cazul în care utilizatorul nu folosește un browser modern, sau unul care poate afisa canvas-urile, atunci o să se afiseze un mesaj, mesaj ce poate fi observat în Figura 5. Tehnologia de tip canvas este compusa în așa fel încât dacă browser-ul este unul care nu are suport pentru canvas, atunci va afisa ceea ce se afla intre aceste tag-uri. Algoritmul pentru a porni jocul se afla într-un fisier extern *Scripts/script.js*. Fișierul de tip JavaScript conține doar 2 funcții principale. Una este pentru a introduce functia *requestAnimationFrame* în pagina,in caz ca nu a fost introdusa, iar cealaltă este o funcție autoapelanta, funcție care se va apela în momentul în care pagina a fost incarcata. În aceasta funcție se afla restul functiilor, funcții care pornesc și modifica logica jocului. Totodata, o alta funcție a acestei funcții autoapelante este acela de a nu lasa variabile globale, astfel ca avem parte doar de variabile locale, definite în cadrul acestei funcții. Acest lucru este folositor pentru a nu avea parte de modificari în cadrul consolei asupra jocului sau a informațiilor jocului, astfel ca utilizatorii nu pot trisa.

Functia *requestAnimationFrame* este folosită pentru a putea afisa jocul la un numar stabil de cadre, în cazul de fata, la 60 de cadre pe secunda. În cazul unor browsere, aceasta funcție este definita în unele moduri diferite, dar în cazul în care avem un browser precum Tor, în care nu poate fi folosită aceasta funcție, avem functia *setTimeout*. Un alt avantaj al acestei funcții, este acela ca nu se va randa nimic în cazul în care fereastra din browser nu este focusata, astfel ca procesorul nu va fi utilizat.

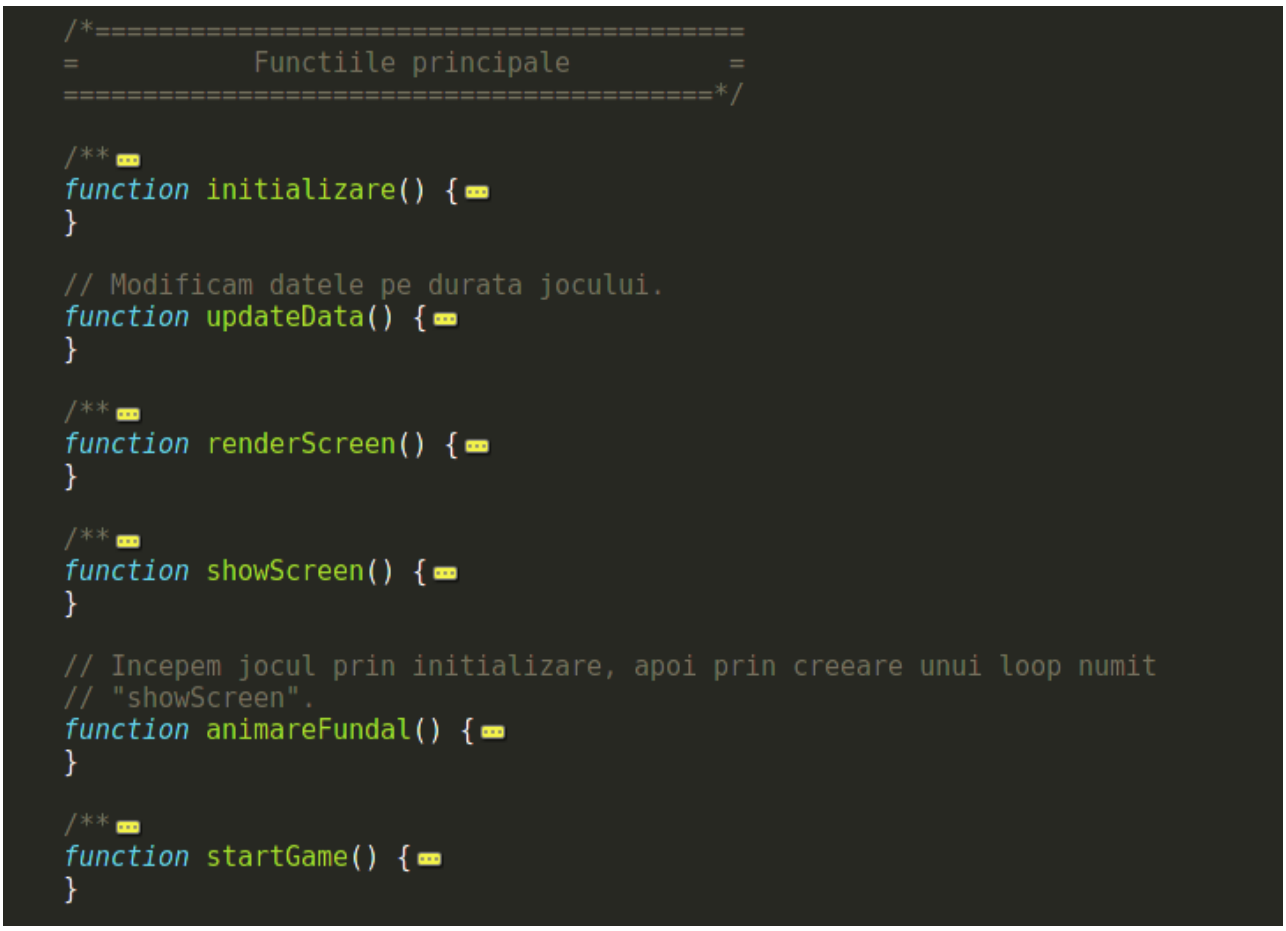


Figura 6.

Apelurile functiilor principale

Functia autoapelativa din JavaScript este de forma: *(function(){/* Algoritm */})()*. JavaScript permite o structura de genul (funcție în cadrul altei functii):

```
function foo() {
        function bar() {
                function test() {
                        /* Algoritm */
                }
                /* ... */
        }
/* ... */
}
```

Așadar, putem introduce mai multe funcții în cadrul functiei autoapelative. Am optat pentru aceasta metoda ca sa nu putem menține toate functiile în cadrul unui singur fisier, funcții care nu pot fi modificate. Totodata fișierul de stocare o să permita ca browser-ul sa retragă doar un singur fisier de la server, fiind mai eficient decât extragerea mai multor fisiere.

Pentru o referinta mai ușoară și pentru a ilustra mai bine ideea de *joc*, folosim un obiect denumit „game”, astfel ca mare parte din parametrii care o să defineasca anumite nivele sau anumite proprietăți ale jocului să fie declarate în cadrul functiei acesteia. În aceasta variabila avem introduse de la început anumite presetari pentru inamici, anumite contoare, precum dimensiunile canvas-urilor (ex. *game.width = 800, game.height = 600*). În aceasta variabila, am ales sa stocam de la bun început contextul de la fiecare canvas folosit. Pentru a putea prelua contextele acestor canvas-uri folosim sintaxa :

```
context = document.getElementById('ID_corespunzator_canvas').getContext(„2d");
```

Înainte de a putea continua avem parte de o mica problema. Deseori se întâmpla ca viteza de internet să fie mai buna decât viteza procesorului sau cea a placii video (depinde de calculator sau de tipul de browser folosit), iar ca sa nu începem adresarea către anumite elemente care în acel moment nu exista, sau nu au fost încă create folosim o funcție Jquery care ne permite a lasa pagina să se incarce, iar pe urma sa ruleze continutul acesteia. Functia Jquery care ne permite acest lucru are o forma de genul:

```
$(document).ready(function () {});
```

Odată incaracata pagina, și continutul acesteia, se poate începe restul algoritmului.

Principiul unui joc

Jocul este format pe principiul algoritmului principal al jocurilor și anume:

- inițializare – funcție care initializeaza informațiile necesare pentru joc
- update – funcție care modifica informațiile în cadrul jocului
- render - funcție care afiseaza jocul
- loop – funcție care se asigura ca jocul va continua, fiind o funcție recursiva care apeleaza functia update, apoi render apoi se reapeleaza
- start – funcție care apeleaza functia de inițializare, apoi functia de loop

Acest principiu de algoritmica este ilustrat în program prin functiile principale (Figura 6), și totodata este dezvoltat puțin, introducand și canteva funcții de ajutor în cadrul acestora, funcții care o să urmeze să fie explicate în continuare.

```
// functie care va incarca imaginile necesare pentru joc
function incarcareImagini(paths) {
    // Retinem lungimea numarului de imagini.
    game.imaginiNecesare = paths.length;
    // Pentur fiecare imagine, o sa folosim algoritmul de initializare si
    // de predefinire a unor imagini in JS.
    for (var i = 0; i < paths.length; i++) {
        var imagine = new Image();
        imagine.src = paths[i];
        game.images[i] = imagine;
        // Daca imaginea curenta s-a incarcat, atunci o sa crestem
        // contorul care ne arata acest lucru.F
        game.images[i].onload =  function(){
            game.imaginiIncarcate++;
        }
    };
}
```

Figura 7

Codul functiei de încărcare a imaginilor

Functia incarcareImagini

Functia start, este apelata în linia 984, dar înainte ei am ales sa incarc imaginile pentru a putea oferii ocazia de a se încarca mai repede. Functia care încarca imaginile se poate observa în figura 7, funcție o să incarce imaginile în cadrul jocului. În apelarea ei, introducem un vector, vector care va conține calea spre aceste imagini. Structura recomandată pentru acesta este de tipul:

```
incarcareImagini([
        "path/to/img1,"
        "path/to/img2,"
        ...
        "path/to/imgN"
]);
```

iar prin acest mod, algoritmul o să poată sa incarce imaginile în ordinea în care i le prezentam. Problema cu aceasta metoda de încărcare, este ca poate dura un timp oarecare în cazul în care avem parte de imagini de dimensiuni mari, deoarece ar trebui sa așteptam un timp oarecum lung. Aceasta problema este ocolita prin functia de pornire a jocului, functia *startGame*.

Algoritmul functiei se bazează pe principiul următor:

- preluam vectorul cu „adresele” imaginilor
- retinem în cadrul variabilei *game.imaginiNecesare* lungimea vectorului
- cu o structura repetitiva de tip *for* incarcam imaginile pe baza unui algoritm relativ de baza pentru JavaScript:
  - declaram o variabila de tip imagine
  - definim sursa imaginii (string-ul introdus în cadrul apelului în vector)
  - variabila va fi stocata pe urma în vectorul *game.images*
  - dacă imaginea a fost incarcata, crestem indicele *game.imaginiIncarcate* cu 1, aratand astfel ca am încărcat o imagine
- continuam structura *for* pana ce terminam toate elementele din vectorul introdus sunt introduse în cadrul vectorul *game.images*

Este de reținut faptul ca aceasta metoda, deși nu prezinta o metoda vizibila de tip *callback,* ea folosește una pentru a permite celorlalte funcții pentru a putea fi apelate.

Functia startGame

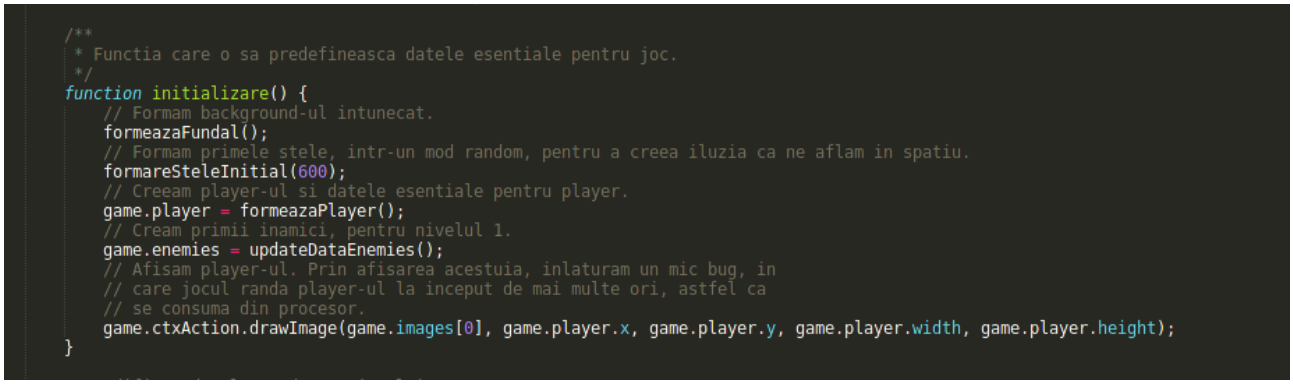
Aceasta funcție este apelata după apelul functiei de încărcare a imaginilor, functia startGame pornește restul jocului, în mod indirect. În mare parte, aceasta funcție verifica dacă imaginile au fost încărcate complet. În ea se verifica 2 opțiuni: prima optiune reprezintă cazul „fericit” în care imaginile au fost încărcate complet, și programul va continua sa ruleze cu restul apelurilor; a doua optiune prezinta cazul cazul în care imaginile nu au fost încărcate, și ca să ne asiguram ca imaginile au fost încărcate complet reapelam functia.

Pentru a verifica dacă imaginile au fost încărcate, folosim 2 variabile declarate înafara functiilor, și anume *game.imaginiIncarcate* și *game.imaginiNecesare*. Aceste variabile sunt explicate în cadrul functiei de încărcare a imaginilor. În cazul în care nu avem imaginile încărcate complet, ajungem pe ramure de tip *else* a structurii de decizie, astfel ca reapelam functia *startGame*, dar nu înainte ca sa afisam un mod de a instiinta utilizatorul ca avem mai de pregătit resurse pentru

```
/**
 * Pe durata incarcarii imaginilor, se va afisa un mesaj 'Loading...'
 */
function renderLoadingScreen() {
    // Predefinirea stilului de text.
    game.ctxBackground.font = "bold 50px Arial";
    game.ctxBackground.fillStyle = "white";
    // Scrierea textului.
    game.ctxBackground.fillText("Loading...", 100, 200);
}
```

Figura 8.

Codul functiei prin care afisam „Loading Screen”



```
/**
 * Functia care o sa predefineasca datele esentiale pentru joc.
 */
function initializare() {
    // Formam background-ul intunecat.
    formeazaFundal();
    // Formam primele stele, intr-un mod random, pentru a creea iluzia ca ne aflam in spatiu.
    formareSteleInitial(600);
    // Creeam player-ul si datele esentiale pentru player.
    game.player = formeazaPlayer();
    // Cream primii inamici, pentru nivelul 1.
    game.enemies = updateDataEnemies();
    // Afisam player-ul. Prin afisarea acestuia, inlaturam un mic bug, in
    // care jocul randa player-ul la inceput de mai multe ori, astfel ca
    // se consuma din procesor.
    game.ctxAction.drawImage(game.images[0], game.player.x, game.player.y, game.player.width, game.player.height);
}
```

Codul functiei de inițializare

joc, prin folosirea functiei *renderLoadingScreen*. În cazul în care imaginile au fost încărcate, o să continuam algoritmul prin apelarea functiei *animareFundal*.

Functia renderLoadingScreen

Aceasta funcție este apelata în timpul verificarii imaginilor, și posibil, pe viitor în cazul în care avem mai multe documente de încărcat pana la inceperea programului. Pentru aceasta funcție, folosim sintaxa de baza pentru a „desena” pe canvas o porțiune de text, și anume:

- definirea stilului de font
- scrierea textului
  - pentru a scrie textul avem nevoie de o sintaxa de forma : *context.fillText(textDeIntrodus, coordonataX, coordonataY);*

În figura 8 puteți observa felul în care avem definita aceasta funcție.

Functia animareFundal

Aceasta funcție nu are un rol așa important, dar ea este necesara. În definirea unui algoritm normal de jocuri, am spus despre functia de tip *start* ca este esentiala, deoarece ea pornește initializarea și loop-ul jocului. În cazul de fata, aceasta este functia de tip *start*.

Prin apelarea acestei funcții o să intram în functia de tip *inițializare*, iar pe urma o să intram în funcție de tip *loop*.

Functia inițializare

Aceasta funcție o să ne folosească pentru a ne permite definirea primului nivel, cât și pentru a forma anumite informații pentru joc. Menționez, ca aceasta funcție nu o să initializeze toate informațiile necesare pentru joc. Va initializa doar informațiile care sunt considerate importante de initializat pentru rularea jocului, fără a se modifica anumite elemente care ar trebui sa rămână constante sau care trebuie configurate pentru a forma jocul în sine.

Functia începe prin configurarea fundalului, formarea stelelor initiale, formarea player-ului, formarea inamicilor pentru primul nivel, și desenarea player-ului pentru prima data.

Configurarea fundalului se executa prin apelarea functiei de formare fundal, *formeazaFundal*, care o să preseteze background-ul cu o culoare neagră. Aceasta conține doar functia de a sterge canvas-ul dedicat background-ului, definirea unui stil de „umplere” drept „black”, și umplerea acestui canvas cu culoarea definita anterior. Codul este valabil în figura 9, iar mai jos aveți o mica explicație asupra acestuia:

context.clearRect(coordonataX_inceput,cootdonataY_inceput,dimensiuneLatime, dimensiuneLungime);

context.fillStyle = ‘culoareDorita’;

context.fillRect(coordonataX, coordonataY, latime, înălțime);

Pentru a forma stelele avem de dezbatut întâi o mica problema. De la începutul proiectului ne gândeam la un mod de a creea iluzia miscarii, un mod prin care am putea sa simulam o mișcare a navei, o mișcare în care nava se mișca către marginea de sus a ecranului la infinit. După un pic de gândire, am venit cu ideea urmatorului algoritm:

- ca sa simulam faptul ca suntem în spațiu avem nevoie de stele. Așadar introducem entitati denumite stele

```
/**
 * Pentru a incepe jocul, este nevoie de a forma fundalul anterior.
 * Functia aceasta formeaza stelele pentru background, formand astfel
 * o iluzie asupra fundalului, de parca ar fi deja in spatiu.
 * @param  {integer} numar -> va stibili numarul de stele care va
 *     aparea la inceputul jocului
 */
function formareSteleInitial(numar) {
    for (var i = 0; i < numar; i++) {
        // In vectorul de stocare al stelelor, introducem pe rand cate o
        // stea, cu informatiile necesare ale ei.
        game.stars.push({
            // Punem pe coorodnata x random
            x: Math.floor(Math.random() * game.width),
            // Punem pe coorodnata y random
            y: Math.floor(Math.random() * game.height),
            // Avem o dimensiune random, de pana la maxim 3 pixeli.
            size: Math.random() * 3
        });
    };
}
```

Figura 10.

Codul pentru formarea *stelelor*

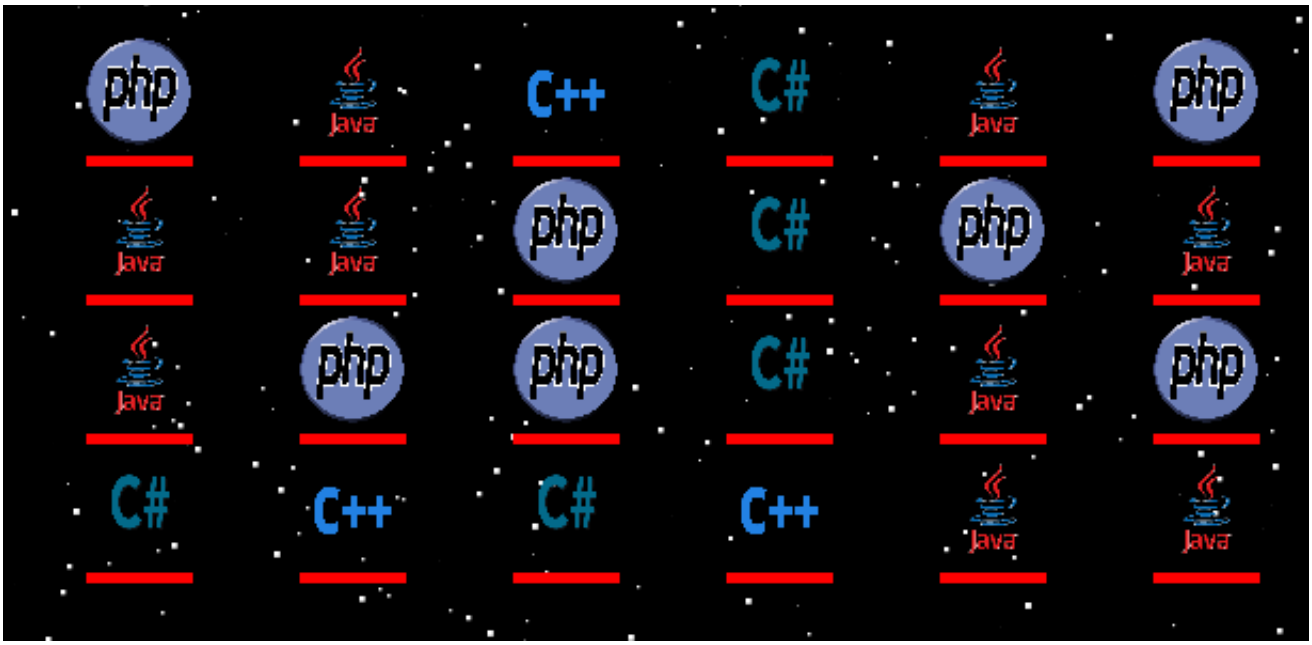


Figura 11.

Așezarea inamicilor

- o să deplasam stelele la fiecare randare, cu o anumita viteza, în cazul de fata cu un pixel pe cadru
- stelele o să vina spre nava utilizatorului, astfel ca o să avem iluzia ca utilizatorul merge *în fata*
- în mod constant apar stele care fac mișcarea continua

Așadar, cu acest algoritm în minte, am început să facem următoarea schema pentru a putea să simulam acest algoritm:

- afisam primele stele – ca sa oferim ideea ca utilizatorul se afla în spațiu
- după ce primele stele sunt afisate, coordonatele y o să scada cu viteza aleasa (1 pixel pe cadru)
- în mod constant o să reapara *stele* ca sa creeze iluzia continuei miscari

Pentru a putea crea acest lucru, în functia de inițializare am introdus functia *formareSteleInitiale*, funcție care o să introduce primele stele în cadrul jocului.

Stelele sunt create după urmatoarele caracteristici: coordonata x, coordonata y și dimensiunea. În functia *formareSteleInitiale* coordonatele sunt puse complet aleatorii cu ajutorul functiei Math.random. Aceasta funcție este făcută în așa fel încât sa formeze un numar complet aleatoriu cuprins intre 0 și 1. Dacă înmulțim acest numar cu dimensiunile ecranului, atunci numărul aleatoriu o să fie cuprins intre 0 și valoarea ce reprezintă dimensiunea ecranului. Astfel, cu aceasta metoda reușim sa formam valori aleatorii pentru stele iar mărimea am ales să fie un numar aleatoriu cuprins intre 0 și 3, astfel ca rezolutia stelelor o să fie de maxim 3 pixeli, iar în momentul când le desenam pe canvas-ul lor (background), o să apara de forma unor patrate. Codul este disponibil în figura 10, prin care ilustram de felul în care am implementat primele coordonate ale stelelor.

Aceste stele sunt pe urma stocate într-un vector denumit *game.stars*, cu informațiile lor necesare, urmând astfel ca să ne fie mai ușor pentru a putea fi mai ușor de accesat sau de a modifica o parte din informațiile lor.

După formarea stelelor, formam player-ul, prin apelarea functiei *formeazaPlayer* care introduce informațiile necesare pentru jucător și nava care îl va reprezenta în cadrul jocului. Aceasta funcție o să introduca informațiile în variabila *game.player.*

Functia aceasta o să formeze coordonatele x și y în funcție de dimensiunile ecrnaului, astfel în momentul actula player-ul se va afla la *coordonata x = game.width * 0.5 – latimea navei*, coordonata y = game.height * 0.8, latimea (width) de game.width * 0.08 și cu înălțimea (height) egala. În felul acesta, imaginea navei va fi de forma patrata. Pe lângă coordonate și dimensiune, avem nevoie de viteza cu care nava se mișca, viața și totodata și proprietatea de mișcare. Aceasta proprietate este necesara pentru a ne putea eficientiza algoritmul. Acest aspect va fi dezvoltat pe în continuarea documentatiei. După determinarea acestor informații, am decis sa le introducem într-o variabila de tip *Object*, iar în final o să returnam acest obiect în variabila dedicata pentru player.

Odată format player-ul, avem nevoie de inamici. Inamicii sunt bazati pe același principiu de coordonate precum player-ul, dar cu anumite proprietăți în plus. Aceștia sunt formați în funcție de nivel, dar cum suntem la functia de inițializare o să dezvaluim felul cum sunt formați pentru acest nivel. Inamicii sunt formați în functia *updateDataEnemies.*

Functia introduce un vector de inamici în cadrul vectorului *game.enemies*. Functia începe prin declararea unui vector enemies. După aceasta declarare avem 2 structuri repetitive de tip *for* care o să formeze inamicii. Aceste structuri sunt așezate asemănător celor pentru structurile de date bidimensionale. Acest lucru este ilustrat în cadrul primului nivel (figura 11). În cadrul nivelelor 2 și 3, aceasta asezare nu are relevanta, deoarece o să

```
// In functie de tipul inamicului, refacem valorile de: hp, hpmax,
// sansa. Fiecare inamic are parte de presetari diferite.
switch(enemy.tip){
    case 0: enemy.hp=10,enemy.hpmax=10,enemy.sansa=0.1; break;
    case 1: enemy.hp=20,enemy.hpmax=20,enemy.sansa=0.2; break;
    case 2: enemy.hp=50,enemy.hpmax=50,enemy.sansa=0.5; break;
    case 3: enemy.hp=30,enemy.hpmax=30,enemy.sansa=0.3; break;
}
```

Figura 12.

Deciderea unor proprietăți ale inamicilor în funcție de tipul lor

```
// Incepem jocul prin initializare, apoi prin creeare unui loop numit
// "showScreen".
function animareFundal() {
    //  Apelam functia de initializare.
    initializare();
    // Dupa initializari, incepem loop-ul jocului.
    showScreen();
}
```

Figura 13,

Functia *animareFundal()*

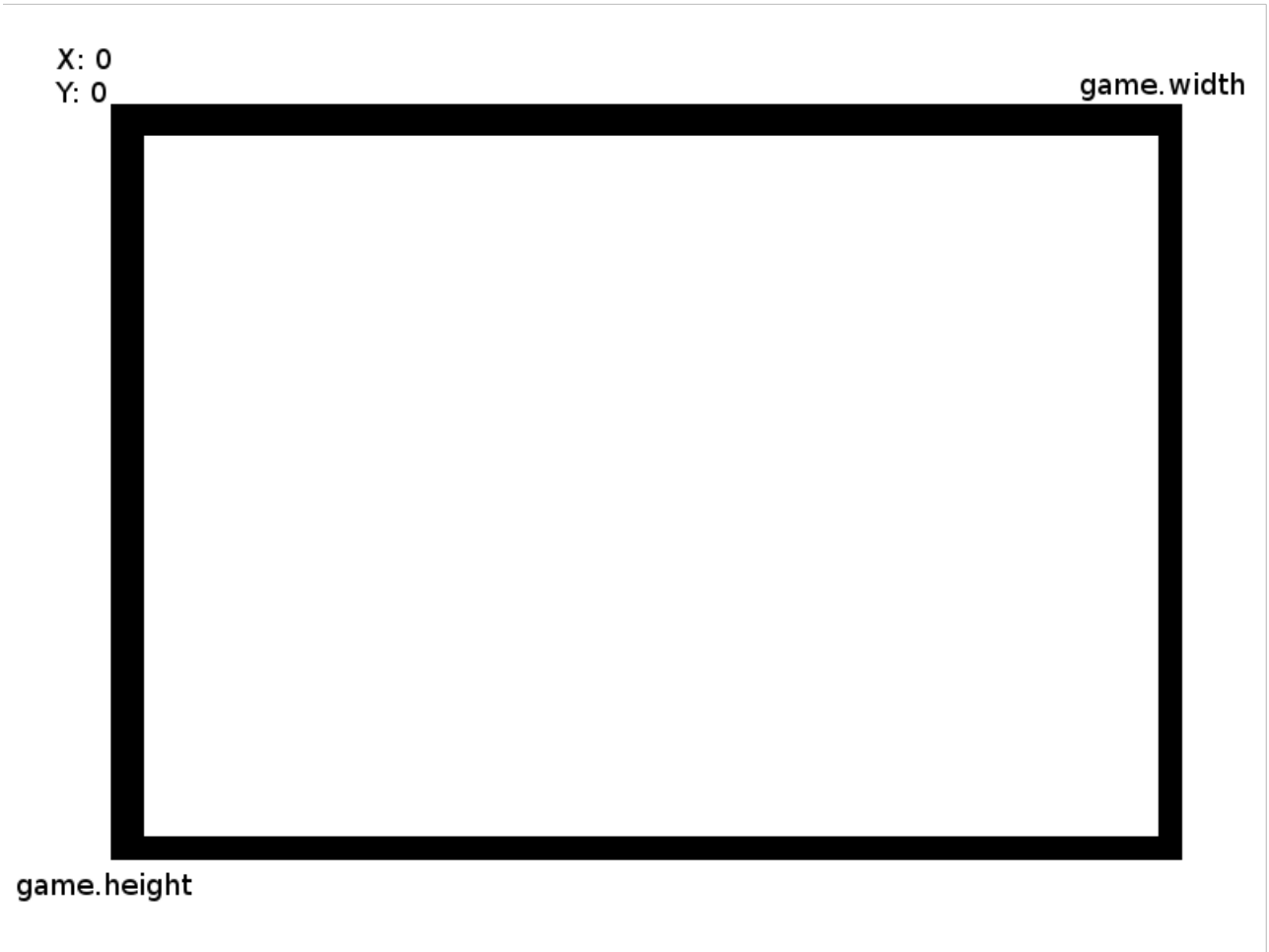


Figura 14.

Felul în care canvas-ul recunoaște coordonatele

primeasca coordonate de tip x și y complet aleatorii, cu conditia sa rămână în spațiul disponibil pe canvas.

Prin cele 2 structuri repetitive, parcurgem prima data coloanele, pe urma liniile. Astfel, putem seta coordonatele x și y în funcție de aceste structuri prin inmultirea indicilor structurilor cu un numar suficient de mare ca sa nu avem suprapuneri asupra inamicilor. În cazul de fata, avem pentru coordonata x latimea inamicului adunata cu indicele de coloana * 100, iar coordonata y este calculata prin inmultirea indicelui de linie cu 65. Pentru a putea arata inamicii, avem nevoie și de latimea și lungimea pe care le-o dedicam inamicilor. Noi am optat ideea de a forma imagini patrate, cu latimi și lungimi egale. Pe lângă aceste proprietăți, avem nevoie și de viața acestor inamici, viteza cu care se mișca, și sa vedem dacă sunt morți sau nu. Înainte de a putea pune variabila de viața, avem nevoie de tipul de inamic, deoarece fiecare inamic în parte are un nivel diferit de viața, și totodata și un nivel diferit de a oferi linii de cod bune. Pentru a ne oferii tipul, am pus imaginile pentru inamici ultimele, astfel ca ele se afla pe pozițiile din vectorul *game.images* pe pozițiile 4, 5, 6, 7; totodata, folosim *Math.random() * 3* pentru a obtine un numar cuprins intre 0 și 3, iar ca sa reprezinte numărul care îl are imaginea îl rotunjim cu functia *Math.round()*. Astfel, putem avea tip 0, 1, 2 sau 3, un tip pentru fiecare fel de inamic disponibil. Pentru a putea pune la fiecare inamic proprietatile sale proprii, folosim o funcție switch pentru a pune proprietatile de hp, hpmax, șansa după cum se poate observa în figura 12, astfel ca o să distribuim limbajelor de programare proprietatile lor descrise la început.

După formarea player-ului, inamicilor și a stelelor, am putea sfârși functia. Din păcate, avem parte de o mica problema, deoarece sunt șanse ca la începutul jocului utilizatorul sa nu fie afisat, și sa rămână invizibil pana în momentul în care se mișca. Pentru a preveni aceasta problema, desenam player-ul pentru moment pe canvas-ul dedicat acestuia.

După terminarea functiei de inițializare, începe functia *showScreen*.

Functia showScreen

Pentru a începe tipul loop al functiei, folosim functia showScreen. Pentru a indeplinii acest loop cu ajutorul functiilor HTML 5, avem folosesc functia definita la început, *requestAnimationFrame*. În figura 13 se poate observa functia *loop* cum este exprimata şi cum se foloseşte de autoapel în functia *requestAnimationFrame*.

Aceasta funcție începe prin apelarea functiei requestAnimationFram, iar ca şi parametru ii oferim o funcție anonima, funcție care o să apeleze functia de tip update, apoi functia de tip render, urmata fiind de către reapelarea recursiva a functiei.

Functia de tip update (updateData)

Aceasta funcție se ocupa cu logica jocului şi modificarea datelor şi informațiilor disponibile. Primul element care îl modifica la fiecare cadru este adaugarea unei noi stele. Putem adauga mai multe stele prin intermediul parametrului trimis, dar am considerat faptul ca avem nevoie doar de o stea la fiecare apelare a functiei.

Pe lângă adaugarea unei noi stele, trebuie sa le și mișcam. Aceasta *mișcare* a stelelor, o realizam prin modificarea coordonatei y, astfel în cazul cazul nostru, stelele se mișca cu o viteza de 1 pixel pe cadru.

Menționez faptul ca coordonatele sunt formatate sub forma prezentata în figura 14, astfel ca o să avem nevoie de a creşte coordonata y ca sa apara impresia de mişcare.

Totuşi, la un moment dat, stelele o să ajungă înafara ecranului, iar ca sa putem sa nu continuam sa calculam coordonatele, eliminam stelele din vectorul *game.stars*, cu ajutorul functiei *array.splice(pozitiaDePeCareStergem, numarElementeDeSters)*. În momentul în care stelele se afla

```
// Functiile JQuery ne ajjuta pentru a gasi exact care tasta este apasata.
// Functie pentru apasarea tastei
$(document).keydown(function(e) {
    game.keys[e.keyCode ? e.keyCode : e.which] = true;
});

// Functie pentru eliberarea tastei.
$(document).keyup(function(e) {
    delete game.keys[e.keyCode ? e.keyCode : e.which];
});
```

Figura 15.

Functiile prin care inregistram tastele apasate

```
<canvas id="background" width="800" height="600"  onkeydown = "keyDown(event)" onkeyup = "keyUp(
event)">
```

Figura 16.

Introducerea acestor funcții în cadrul paginii

```
/**
 * Modificam miscarea player-ului in functie de tastele apasate.
 */
// Daca cumva am apasat tasta spre stanga, sau tasta 'a',
if (game.keys[37] || game.keys[65]) {
    // Si daca cumva nu iesim inafara ecranului,
    if(game.player.x > 0) {
        // Modificam coordonata 'x' a jucatorului.
        game.player.x -= game.player.speed;
        // Modificam proprietatea de tip 'miscare' pentru a putea
        // afisa player-ul.
        game.player.miscare = true;
    }
};
```

Figura 17.

Algoritmul prin care mișcam nava utilizatorului spre stânga

la o coordonata y mai mare decât lungimea ecranului, în cazul de fata *game.width + 10*, atunci o să le eliminam cu ajutorul acestei funcții.

Am modificat datele pentru background, acum urmează sa modificam datele pentru player. Una dintre cele mai greu de înțeles idei la început, a fost felul în care aș putea sa inregistrez în browser tastele apasate, astfel ca utilizatorul se va mișca. După puțin timp de gândire și puțin ajutor din partea site-ului www.StackOverflow.com, am găsit o metoda de a putea inregistra tastele.

Algoritm pentru inregistrarea tastelor

Pentru a inregistra tastele am folosit un vector dedicat retinerii tastelor, vector denumit *game.keys*. Acest vector este va reține variabile de tip bool pe pozițiile sale. Cu ajutorul a 2 funcții dedicate unui canvas, prin care punem codul tastelor apaste în vector. Avem nevoie de 2 funcții pentru a le reține. Aceste funcții sunt disponibile în figura 15, iar apelarea acestora se poate vedea în figura 16.

Fiecare tasta are un cod unic, cod care este exprimat prin numere întregi cuprinse intre 0 și 256, în funcție de tastatura, dar valorile tastelor dedicate cifrelor, literelor și tastelor speciale rămân constante. Astfel, dacă știm codul acestor taste, putem sa le retinem, și doar în care aceste coduri au fost apasate pe tastatura, aplicam algoritmul dedicat acestora.

În momentul în care o tasta este apasata, ea primește în cadrul functiei *keydown*, se preia codul tastei apasate, iar pe urma, codul acesteia este preluat și poziția pe care o exprima acest cod de tasta va fi folosit în expresia prezentata în figura 15 pentru a pune în *game.keys[codTastaApasata]* valoarea de *true*, astfel ca aratam ca exista tasta apasata în vector. În momentul în care o tasta este eliberata după ce a fost apasata, avem a doua funcție *keyup*, funcție care pune în *game.keys[codTastaEliberata]* cu valoarea de *false*, astfel ca aratam ideea ca tasta nu mai este apasata.

Tastele care le retinem sunt: 4 de directii, una de tragere. Cele 4 de direcție ar putea fi doar 2, dar deseori, utilizatoriii poate doresc sa folosească săgeţile, sau poate doresc sa folosească *a* sau *d*, pentru mișcare. Mișcările posibile pentru utilizatoru sunt fie spre stânga, fie spre dreapta.

Deoarece folosim doar aceste taste, avem nevoie de reținerea a 5 coduri:

- tasta *a* - pentru a ne mișca spre stânga (cod 65)
- tasta *d* – pentru a ne mișca spre dreapta (cod 68)
- săgeata spre stânga – pentru a ne mișca spre stânga (cod 37)
- săgeata sprea dreapta – pentru a ne mișca spre dreapta (cod 39)
- spațiu – pentru a trage (cod 32)

Mișcarea utilizatorului

Când este apasata o tasta dedicata miscarii spre stânga, scădem din coordonata x viteza navei, pe durata miscarii pe care o avem (adică pe durata apasarii tastelor). În anumit situații jucătorul o să iasă înafara ecranului, dar ca sa prevenim aceste situații, verificam conditia dacă cumva o să ieșim înafara cadrului, dacă cumva se modifica poziția jucatorului. Aceasta conditie este evidentiata în cadrul figurii 17. Dacă utilzatorul se mișca, o să ii oferim proprietatii de *mișcare* valeaarea de *true*, astfel ca aratam ca utilizatorul s-a mișcat. Dacă player-ul a fost mișcat, atunci o să afisam în functia de render noua poziție a jucatorului. În mod analog, același algoritm este și pentru mișcarea spre dreapta.

```
// In momentul in care apasam tasta 'space', si putem trage,
if (game.keys[32] && game.contorInitialProiectil <= 0 && canShoot == true) {
    // Incarcam un nou proiectil
    game.proiectilPlayer.push({
        // Coordonata 'x' va incepe din mijlocul player-ului
        x: (game.player.x + (game.player.width / 2) - 5),
        // Coordonata 'y' incepe din varful imaginii, mai exact din
        // zona 'de tun' a imaginii
        y: (game.player.y + 10),
        // Latimea si lungimea imaginii le pun identice, pentru a
        // putea definii o imagine patrata.
        width: 10,
        height: 10,
        // Viteza proiectilului
        speed: 9,
        // ID-ul imaginii
        image: 1
    });
    // Resetam contorul de tragere.
    game.contorInitialProiectil = game.contorFinalProiectil;
};

// Pentru a putea trage, avem un contor care nu ne lasa sa tragem
// decat in mommentul in care a trecut o parte de din cadre. La
// fiecare cadru se scade cu unu contorul.
if (game.contorInitialProiectil > 0) {
    game.contorInitialProiectil--;
};
```

Figura 18.

Introducerea proiectilului utilizatorului în cadrul jocului

Functia de tragere pentru utilizator

În momentul în care se apasa tasta de spațiu, se verifica dacă putem trage. Aceste verificări sunt necesare pentru a nu lasa utilizatorul sa abuzeze de lansarea proiectilelor.

Verificarile folosite sunt formate pe baza unui contor improvizat și prin setarea variabilei *canShoot*, variabila de tip bool, punem proprietatea ca jucătorul sa poată sa tragă doar în cazul în care are voie. Contorul improvizat se bazează pe o mecanica simpla dar eficiența:

- se fixeaza o valoare maxima pentru contor, și o variabila care se va incrementa
- se incrementeaza cu 1 la fiecare revenire în algoritm
- în momentul în care variabila ajunge să fie divizibila cu valoarea maxim (se poate spune și dacă este mai mare sau egala decât aceasta), atunci o să resetam contorul

Algoritmul pentru tragere are nevoie de acest contor pentru ca utilizatorul sa nu abuzeze de lansarea de proiectile. În momentul în care conditiile sunt indeplinite, atunci utilizatorul poate sa tragă un proiectil.

Proiectilul este un obiect cu proprietatile:

- coordonata x
- coordonata y
- latime/lungime
- viteza
- imagine (ID-ul imaginii din vectorul *game.images*)

Aceste proprietăți sunt relativ fixe, cu excepția coordonatei x, deoarece ea va fi pozitionata în *centrul* navei. Proiectilul va fi *impins* direct în vectorul *game.proiectilPlayer*, astfel ca fiecare proiectil o să fie reținut într-un vector.

Informațiile proiectilului și algoritmul pentru tragere sunt disponibile în figura 18.

Pentru a mișca proiectilele folosim un algoritm asemănător cu cel al stelelor, modificand doar coordonata y, adaugand viteza la coordonata y. În cazul în care, după ce proiectilul a ajuns înafara ecranului, îl eliminam din vector cu ajutorul functiei *vector.splice()*.

Inamici. Mișcarea inamicilor

Inamicii au un anumit tipar de mișcare în fiecare nivel. La nivelul 1, ei nu se mișca deloc. La nivelul 2, ei au o mișcare de sus-jos de-a lungul axei y, iar la nivelul 3 avem fie o mișcare stânga-dreapta, fie o mișcare identica cu cea de la nivelul 2. Mișcarea acestor inamici este realizata prin modificarea coordonatei y sau a coordonatei x, în funcție de nivel sau de tipul de mișcare pe care îl au. Pentru a putea creea ideea unei mișcari repetitive, am venit cu ideea unui alt contor improvizat, asemănător cu cel pentru lansarea de proiectile ale utilizatorului. Mișcarea inamicilor este realizata cu ajutorul urmatorului algoritm:

- cream un indice *game.contorInamici*
- cream o constanta în care avem valoarea maxim a contorului de inamici, *game.contorTimpMaximInamici*
- incrementam *game.contorInamici*

```
// Daca a trecut un oarecare timp de miscare,
if (game.contorInamici % game.contorTimpMaximInamici == 0) {
    // Punen opusul directiei de mers
    game.deplasareInamicStanga = !game.deplasareInamicStanga;
};

// Daca ne aflam in cadrul nivelului 2, miscarea va fi de tip sus/jos,
// astfel ca modificam coordonata de tip 'y'
if(level == 2){
    // Pentru fiecare inamic existent,
    for(var i in game.enemies){
        // Daca contorul de deplasare este pozitiv,
        if (game.deplasareInamicStanga) {
            // Scadem din coordonata 'y' viteza de miscare
            game.enemies[i].y -= game.enemySpeed;
        } else {    // In mod contrar
            // Crestem din coordonata 'y' viteza de miscare
            game.enemies[i].y += game.enemySpeed;
        };
    }
}

// Daca ne aflam la nivelul 3, miscarea este oarecum aleatorie. Pentru
// aceasta, avem nevoie de tipul miscarii inamicului.
if(level == 3){
    // Pentru fiecare inamic,
    for(var i in game.enemies){
        // Verificam de miscare.
        // Daca inamicul are un tip de miscare '1'
        if(game.enemies[i].movement == 1){
            // Modificam coordonata 'y'
            if (game.deplasareInamicStanga) {
                game.enemies[i].y -= game.enemySpeed;
            } else {
                game.enemies[i].y += game.enemySpeed;
            };
        } else {    // Altfel modificam coordonta de tip 'x'
            if (game.deplasareInamicStanga) {
                game.enemies[i].x -= game.enemySpeed;
            } else {
                game.enemies[i].x += game.enemySpeed;
            };
        }
    }
}
```

Figura 19,

Codul destinat miscarii inamicilor

```
/**
 * Functia de coliziune se bazeaza pe obtinerea coordonatelor ale unor
 * obiecte. Verificam daca coliziunea se intampla in unul dintre cele 4
 * cazuri
 * @param  {object} obiectUnu Primul obiect care contine proprietatile de
 *      tip: x, y, width, height
 * @param  {Object} obiectDoi Al doilea obiect care contine proprietatile
 *      de tip: x, y, width, height
 * @return {Boolean}           Ne spune daca avem parte de o coliziune
 */
function coliziune(obiectUnu, obiectDoi) {
    return ( // Returnam direct valoarea expresiei in caz ca avem o coliziune.
        obiectUnu.x < obiectDoi.x + obiectDoi.width     &&
        obiectUnu.x + obiectUnu.width > obiectDoi.x     &&
        obiectUnu.y < obiectDoi.y + obiectDoi.height    &&
        obiectUnu.height + obiectUnu.y > obiectDoi.y
    );
}
```

Figura 20.

Codul functiei de coliziune

- cream variabila *game.deplasareInamicStanga* – aceasta variabila stabileste indirect direcția în care se mișca inamicii
- în momentul în care game.contorInamici este divizibil cu variabila constanta, inversam variabila (*game.deplasareInamicStanga = !game.deplasareInamicStanga)*
    - în felul acesta, dacă cumva variabila va avea valoarea *true,* atunci în urma unei condiții indeplinite, o să primeasca valoarea de *false* și viceversa
- în funcție de valoarea variabilei *game.deplasareInamicStanga*, coordonata x, respectiv y, va crește cu viteza inamicului în funcție de viteza sa dacă valoarea este false. Dacă este *true*, atunci aceste coordonate o să scada cu viteza pe care o au

Pentru a putea mișca inamicii, adaugam/scadem viteza pe care o au, în funcție de variabila *game.deplasareInamicStanga.* În momentul în care aceasta variabila este schimata, atunci inamicii o să își schimbe direcția de mișcare.

Codul disponibil pentru acest algoritm este disponibil în figura 19. După cum se poate observa în cod, avem variante de mișcare în funcție de nivel. Aceste nivele au diferite metode a de mișca inamicii.

Coliziunea

Am prezentat felul cum utilizatorul se mișca, felul cum inamicii se mișca, și proiectilele utilizatorului se mișca. Odată ce avem aceste elemente, putem începe verificarea unor coliziuni. Înainte de a prezenta algoritmul de coliziune trebuie sa înțelegem ideea ***hitbox-urilor***. Aceste hitbox-uri reprezintă zona în care se afla imaginile. În cazul de fata, hitbox-ul este identic cu mărimea imaginilor. Pentru un calcul mai simplu pentru procesor, și datorită unor conventii, aceste hitbox-uri reprezintă un patrat în jurul entitatii, la care retinem urmatoarele variabile:

- coordonata x
- coordonata y
- latimea
- lungimea

Coliziunea a 2 obiecte reprezintă suprapunerea a 2 hitbox-uri. Aceste suprapuneri sunt descrise în codul din figura 20. Functia *coliziune* primește 2 parametrii, parametrii care reprezintă obiecte de tip javascript, care necesita proprietatile *x, y, width, height*.

Pentru o mai simpla înțelegere, și pentru ca explicatia să fie asemănătoare cu codul din figura 20, cei 2 parametrii o să fie denumiti obiectUnu, obiectDoi.

Functia o să verifice suprapunerea acestor hotbox-uri, prin verificarea a 4 condiții:

- obiectUnu.x < obiectDoi.x + obiectDoi.width - dacă coordonata x a primului obiect este mai mica decât cea x a primului obiect adunata cu latimea acestuia
- obiectUnu.x + obiectUnu.width > obiectDoi.x – dacă coordonata x a primului obiect, adunata cu latimea acestuia este mai mare decât coordonata x a celui de-al doilea obiect
- obiectUnu.y < obiectDoi.y + obiectDoi.height – dacă coordonata y a primului obiect este mai mica decât coordonata y a celui de-al doilea obiect adunata cu lungimea acestuia
- obiectUnu.height + obiectUnu.y > obiectDoi.y – dacă coordonata y a primului obiect, adunata cu lungimea acestuia, este mai mare decât coordonata y a celui de-al doilea obiect

```
// Pentru fiecare inamic,
for (var contorInamic in game.enemies){
    // Pentru fiecare proiectil,
    for (contorProiectil  in game.proiectilPlayer){
        // Daca exista o coliziune intre proiectilul curent si player
        if (coliziune(game.enemies[contorInamic], game.proiectilPlayer[contorProiectil])) {
            // Scadem viata inamicului cu 10 (damage-ul proiectilului de la player)
            game.enemies[contorInamic].hp -= 10;
            // Daca cumva viata este 0,
            if(game.enemies[contorInamic].hp == 0){
                // Punem proprietatea inmicului '.mort' drept true.
                game.enemies[contorInamic].mort = true;
            }
            // Stergem inamicul curent de pe ecran distrus.
            game.ctxBullet.clearRect(game.proiectilPlayer[contorProiectil].x,
                game.proiectilPlayer[contorProiectil].y,
                game.proiectilPlayer[contorProiectil].width + 5,
                game.proiectilPlayer[contorProiectil].height + 10);
            // Il eliminam din vectorul de inamici.
            game.proiectilPlayer.splice(contorProiectil, 1);
                // Continuam cu algoritmul.
                continue;
        }
    }
}
```

Figura 21.

Dacă toate aceste 4 condiţii sunt indeplinite, atunci înseamnă ca avem o coliziune intre aceste 2 obiecte. Prin returnarea acestei condiţii, putem returna fie valoare *true*, fie valoarea *false.* Astfel, prin apelarea acestei funcţii, putem verifica în cadrul unei structuri de decizie coliziunea.

Coliziunea dintre proiectile și inamici

Acum ca avem functia de coliziune, putem verifica dacă inamicii şi proiectilele utilizatorului sunt în coliziune. În momentul în care ele se afla într-o coliziune, scădem viata inamicul aflat în coliziune, şi eliminam proiectilul care a nimerit acest inamic.

Pentru a putea verifica fiecare coliziune, avem nevoie de o structura repetitiva atât pentru proiectile, cât şi pentru inamici. Pentru fiecare inamic în parte, o să verificam dacă se afla în coliziune cu un proiectil. Dacă se afla, o coliziune, viaţa inamicului o să scada cu 10 puncte, iar proiectilul o să fie eliminat din vectorul sau prin intermediul functiei *vector.splice().* Totodata, stergem în mod direct proiectilul de pe ecran (deşi nu îşi găseşte locul în aceasta funcţie, am ales sa punem aici functia pentru a evita anumite probleme în browser-ul FireFox, browser care o să ajungă sa apeleze în anumite condiţii de 2 ori functia de tip *update* pana o să ajungă sa apeleze functia de tip *render,* aşadar am ales sa prevenim aceste probleme prin punerea acestei funcţii în cadrul acestei strucuti).

Dacă cumva inamicul actual ajunge sa aibe viaţa egala cu 0, atunci o să ii definim proprietatea *.mort* cu valoarea true (figura 21), iar pe urma acesta va fi eliminat din cadrul jocului după trecerea timpului indicat de proprietatea *.timpMort.*

Proiectilele inamicilor

În jocuri de acest tip, avem nevoie ca inamicii sa poată sa tragă. În mod normal, inamicii sunt aleşi într-un mod aleatoriu ca sa poată sa tragă, astfel ca inamicii o să aibe ocazia sa tragă la fiecare apelare a functiei. Noi am optat pentru o alta metoda. Metoda noastră se bazează pe un alt timer improvizat, care în funcţie de nivel, la un oarecare timp, mai mulţi inamici o să tragă deodată.

Algoritmul pe care îl avem este următorul:

- parcurgem fiecare inamic în parte
- dacă ne aflam la un nivel la care nu avem un boss (în cazul de fata, nivelul 4)
  - inamicul actual la care suntem, ca sa poată sa tragă, are o şansa de 20%
  - dacă are şansa sa tragă, atunci verificam dacă este momentul ca să se tragă. La fiecare nivel, avem un timp diferit pentru a trage. De exemplu: la nivelul 1 trebuie sa treacă 180 de cadre (3 secunde dacă jocul o să ruleze la 60 de cadre pe secunda), la nivelul 2 trebuie sa treacă 60 de cadre (o secunda) etc. şi dacă se poate trage se continua algortimul
  - se intra în functia de inamicTrage prin care se introduc coordonatele x, y şi şansa inamicului care trage
    - aceasta funcţie este foarte asemănătoare cu functia pentru tragere a proiectilelor jucatorului, singura diferenţa este ca va diferi în funcţie de imaginea proiectilului şi de şansa pe care o are inamicul. Tipul de proiectil are aceleaşi principii, dar imaginea (ID-ul imaginii) decide mare parte din proprietatile pe care le are.
    - Functia prestabileste locatiile proiectilului, viteza şi dimensiule sale
    - în funcţie de tipul inamicului, fiecare apel al functiei are şansa de a lansa o linie de cod buna, o linie de cod care o să ajute utilizatorul sa indeparteze din bug-uri

        - dacă nu își *nimereste* șansa, atunci o să urmeze să i se aloce o linie de cod rea, fiind o șansa de 50%-50% pentru aceasta (dar doar în cazul în care nu avem o linie de cod buna). Aceasta linie de cod rea este stabilită de la început drept imaginea cu id-ul 9 (o linie de cod rea), iar pe urma, are șansa să se transforme într-o linie de cod buna. În caz contrar, are șansa de 50% sa devină o alta linie de cod rea. Astfel adaugam o mica diversitate în cadrul jocului
        - noul proiectil este *impins* în vectorul *game.proiectilInamici* prin intermediul functiei *vector.push(element)*
    - acest algoritm este identic pentru toate 3 nivele, astfel ca modificam doar timpul de „tragere”
- resetam valoare contorului la 0
- Dacă suntem la invelul 4, algoritmul rămâne aproape identic, singurele modificari sunt timpul de tragere, damage-ul proiectilelor inamicilor, și faptul ca proiectilele apar într-un mod random de dedesuptul navei *boss*

Scorul și calcularea acestuia

Pentru a putea calcula scorul ne folosim de logica descris la început. În cazul în care utilizatorul a fost lovit de o linie de cod rea, atunci o să adaugam 10 bug-uri, altfel, dacă am fost loviți de o linie de cod buna, atunco scădem 10 bug-uri.

În cadrul functiei de tip update, avem introdusa în restul programelor aceasta sintaxa, astfel ca scorul este modificat la fiecar coliziune cu proiectile. În cazul în care după calcule, scorul va ajunge negativ, atunci scorul nu va fi modificat. În cazul în care scorul a ajuns mai mare sau egal cu 100, atunci se va intampina mesajul *Ai pierdut!...* fiind urmat de sfârșitul jocului.

Finalul jocului

La finalul functiei exista o structura de decizie, o strucura care verifica dacă jocul s-a terminat. Consideram jocul terminat în momentul în care toți inamicii au fost eliminați din cadrul jocului. Metoda prin care verificam dacă jocul a fost terminat este prin verificarea lungimii vectorului de stocare a inamicilor. Dacă lungimea lui este mai mare decât 0, atunci se mai poate continua jocul. Altfel se trece peste aceasta etapa.

Functia finnishLevel

Prin aceasta funcție, crestem nivelul. În funcție de nivelul la care suntem, o să incarcam setarile pe care le avem pentru fiecare nivel în parte, cum ar fi damage-ul pentru fiecare nivel și numărul de inamici, sau introducerea în nivelul 4 a boss-ului.

În urma acestei funcții, mereu se va reface vectorul de stocare al inamicilor cu noile specificatii ale acestora.

Functia de tip render

Aceasta funcție se ocupa cu afisarea pe ecran a graficii jocului.

Pentru a putea afisa pe canvas, folosim sintaxa specifica HTML 5:

- pentru curatarea ecranului se folosește:
    - context.clearRect(x, y, width, height);
- pentru a umple fundalul cu o culoare folosim

```
// Salvam ultimele desene ale inamicilor cu informatiile
// necesare, deoarece urmeaza sa pune 'hp-bar' la fiecare
// inamic.
game.ctxInamici.save();
// Hp-bar-ul o sa fie de culoare rosie, asa ca ii punem
// proprietatea de culoare necesara.
game.ctxInamici.fillStyle = "red";
// Desenam hp-bar
game.ctxInamici.fillRect(inamic.x,inamic.y + inamic.height + 5, inamic.width * (inamic.hp/inamic.hpmax), 5);
// Restituim proprietatile salvate anterior.
game.ctxInamici.restore();
```

Figura 22.

- context.fillStyle = ‘culoareaAleasa’

- pentru a afisa o imagine folosim:
  - context.drawImage(imagine, x, y, width, height);
- pentru a afisa o culoare folosim:
  - context.fillRect(0, 0, width,height);
- pentru a putea pune optiunile pentru text:
  - context.fillStyle = ‘proprietati de text’;
- pentru a afisa textul pe canvas folosim:
  - context.fillText(mesaj, x, y);
- pentru a salva și restitui ultimele setari stilistice pentru acel canvas folosite anterior folosim sintaxa:
  - context.save(); - pentru a salva ultimele setari de stil
  - context.restore(); - pentru a restitui ultimele setari de stil

Desenarea unei entitati se bazează pe curatarea canvas-ului în jurul acesteia, iar pe urma desenarea entitatii (fie imaginea, fie porţiunea, fie textul). Fiecare inamic, proiectil, stea și utilizatorul reprezintă o entitate și li se aplica aceasta idee. Toate entitatile nedescrise în continuare folosesc aceeași metoda, iar în cele ce urmează o să descriu exact anumite probleme intampinate, sau anumite idei mai unice.

Desenarea player-ului

Pentru a desena player-ul verificam dacă acesta s-a mișcat, dacă cumva merita să se reafiseze acesta, pentru a econimisi resursele necesare. După redesenarea acesteia, proprietatea de *mișcare* primește valoarea de false. Apoi, este urmat de algoritmul de desenare a unei entitati, descris anterior.

Desenarea inamicilor

Același algoritm se aplica și acestora, precum și la restul entitatilor, singura diferența fiind desenarea hp-bar-ului. Aceasta metoda de a afisa viața inamicilor a fost preluata de pe forumurile w3school.com, în care se afisau mai anumite metode de infrumusetare a unui canvas, sau metode a imbogati aspectul pe care îl adoptam.

Codul este disponibil în figura 22.