HTML / Javascript - Tutorial "Life": il gioco della vita

Scopo del presente tutorial è di realizzare un versione HTML / Javascript del **"gioco della vita"** o *life* del matematico inglese John Conway. Si tratta del più famoso esempio di automa cellulare: un modello matematico usato per descrivere l'evoluzione di sistemi complessi. Si rimanda alla vasta bibliografia sull'argomento per maggiori dettagli.

Qui basta sapere che nel gioco della vita si simula l'evoluzione di un insieme di cellule poste in una griglia bidimensionale. Ad ogni istante di tempo lo stato di una cellula dipende dallo stato delle celle che la circondano. I dettagli li scopriremo via via che andiamo avanti nella realizzazione.

Passo 1: creare una griglia con utilizzando un canvas

In questo primo passo scriveremo una pagina contenente una griglia di quadrati che costituirà la base per il nostro gioco. Per prima cosa, tramite la pagina web, chiederemo la dimensione della griglia, poi la griglia sarà costruita con le dimensioni date utilizzando un **Canvas**.

Inoltre in questa fase diamo l'impostazione al nostro progetto utilizzando file separati: uno per il foglio di stile (CSS), uno per rappresentare la pagina HTLM e uno con il codice Javascript. Creiamo quindi tre file:

- `life.css`: file con la definizione di stile. Contiene solo i colori di background e foreground da assegnare a tutti gli oggetti. Per la scelta dei colori abbiamo preso spunto al tema "Solarized".

```
body {
    background-color: #073642;
                color: #93a1a1;
}
```

Potevamo integrarlo all'interno del file HTML, ma lasciarlo fuori ha dei vantaggi che scopriremo con l'esperienza. In generare è una buona idea separare il contenuto della pagina (file HTML) dai dati che utilizziamo per realizzarla (il file CSS) e dal codice che costituisce il motore che fa evolvere la pagina (il codice Javascript).

- `life.html`: file con la definizione pagina HTML. Lo snippet di codice riportato sotto contiene la pagina intera, con commenti per spiegare il significato delle varie sezioni. Come si vede, questa pagina include il file `life.js` che vedremo al prossimo punto e che conterrà il codice javascript che realizzerà il gioco.

```
<!DOCTYPE html>
<html lang="it">
<head>
  <meta charset="utf-8"/>
  <!-- include lo stile sheet -->
  <link href="life.css" rel="stylesheet" type="text/css">
</head>

<!-- include lo script -->
<script type="text/javascript" src="life.js"></script>

<body>
  <!-- Questa e' l'area dove inizialmente sono specificati i parametri
       del gioco. Quando si premera' il tasto "Comincia" quest'area
       verrà nascosta e verrà visualizzata l'area successiva -->
  <div id="setup">
    <p>
      Altezza griglia (numero righe):
```

```
        <input type="text" id="altezza" size="4" value="20">
    </p>
    <p>
      Larghezza griglia (numero colonne):
       <input type="text" id="larghezza" size="4" value="20">
    </p>
    <p>
      <input type="button" id="VIA" value="Comincia"
        onclick="eseguiProgrammaLife(document.getElementById('altezza').value,
        document.getElementById('larghezza').value);">
    </p>
  </div>

  <!-- Questa è l'area di gioco vera e propria, il cui contenuto
       sara' costruito dalla funzione js preparaCanvas() -->
  <div id="lifegameBoard">
  </div>
</body>
</html>
```

Come si vede, il `body` della pagina è diviso in due parti: un `<div>` di setup che visulizza i campi per introdurre il numero di celle della griglia (righe e colonne della matrice di gioco) e il pulsante per dare il via alla simulazione e un `<div>` principale inizialmente vuoto che verrà riempito con la griglia una volta premuto il bottone "Comincia". Questa pagina, come il file HTML non cambierà più per il resto del tutorial. Concentriamoci adesso sul vero motore del gioco.

- `life.js`: è file con il codice Javascript. In questa prima fase il codice semplicemente viene attivato alla pressione del tasto "Comincia". Si entra nella funzione `eseguiProgrammaLife()` che per il momento si limita a nascondere il `<div>` di setup e poi a costruire la griglia.

La griglia viene costruita tramite un Canvas HTML. I Canvas sono utilizzati proprio per fare grafica "al volo" su una pagina HTML. Le operazioni grafiche sono realizzate tramite funzioni javascript.

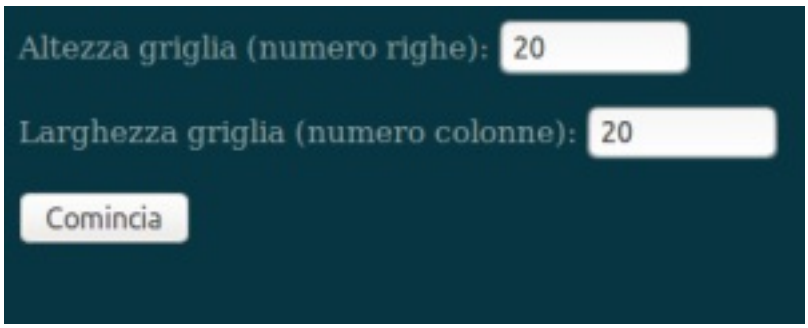
Ecco la prima versione del file `life.js`:

```
// Variabili globali
var c = document.createElement("canvas");
var ctx = c.getContext("2d");
var cellsize = 10;

var numeroRighe;
var numeroColonne;

// ------------------------------------------------------------------------
// Programma principale chiamato alla pressione del pulsante "Comincia"
//  per il momento semplicemente nasconde i campi per il setup e disegna il
// campo vuoto
function eseguiProgrammaLife(nrows, ncols) {
  // salva il numero di righe e di colonne in due variabili
  // globali che saranno utilizzate nel seguito dal resto del Programma
  numeroRighe = parseInt(nrows);
  numeroColonne = parseInt(ncols);

  // nasconde i campi per la lettura delle dimensioni del campo
  MostraNascondi('N');

  // disegna il campo di gioco
  preparaCanvas();
}

// ------------------------------------------------------------------------
```

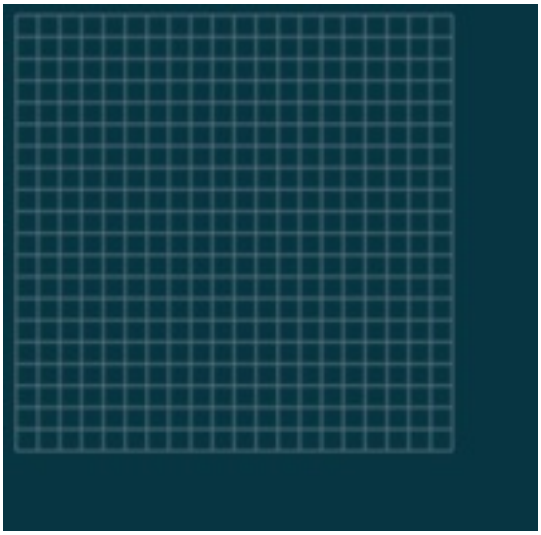
```
// Serve per mostrare e nascondere gli item per inserire
// le informazioni del campo da creare
function MostraNascondi(x) {
  if (x=='N') document.getElementById('setup').style.display='none';
  else document.getElementById('setup').style.display='block';
}

// ------------------------------------------------------------------------
// Disegnol campo con le dimensioni date
function preparaCanvas() {
  div = document.getElementById("lifegameBoard");
  c.width = (cellsize + 1) * numeroColonne + 2;
  c.height = (cellsize + 1) * numeroRighe + 2;

  ctx.beginPath();
  disegnaCampo("#657b83")
  ctx.stroke();
  div.appendChild(c);
}

// ------------------------------------------------------------------------
function disegnaCampo(color) {
  var cellfullsize = cellsize + 1;
  var sizetotalx = cellfullsize * numeroColonne;
  var sizetotaly = cellfullsize * numeroRighe;

  for (var row = 0, currenty = 1; row <= numeroRighe; row++, currenty +=
cellfullsize)
  {
    ctx.moveTo(1, currenty);
    ctx.lineTo(1 + sizetotalx, currenty);
  }
  for (var col = 0, currentx = 1; col <= numeroColonne; col++, currentx +=
cellfullsize)
  {
    ctx.moveTo(currentx, 1);
    ctx.lineTo(currentx, 1 + sizetotaly);
  }
  ctx.strokeStyle = color;
}
```

Caricando la pagina `life.html` con un browser, il risultato sarà una prima pagina con i controlli per inserire i parametri della simulazione (la dimensione della griglia) e il pulsante "Comincia" che lancia lo script Javascript. La seconda pagina è il risultato della manipolazione dello script: i controlli scompaiono e compare la grid di gioco. Quello che appare è riportato nelle immagini che seguono.

La schermata con i controlli:



La griglia vuota:



Passo 2: riempimento della griglia di gioco con cellule vive

Abbiamo detto che il gioco della vita simula l'evoluzione di un insieme di cellule in base ad alcune regole che vedremo successivamente.

Quello che ci serve in questa fase è preparare le strutture dati che conservano lo stato del nostro campo. Per questo ci servirà una matrice bidimensionale di dimensione pari al numero di righe per il numero di colonne del nostro campo da gioco. Se un item di questa matrice contiene il valore "1" (l'intero uno) vuol dice che in quella posizione c'è una cellula viva, altrimenti contiene 0 (zero) e vuol dire che in quella posizione non c'è una cellula.

Successivamente coloreremo il campo da gioco in modo da rispecchiare il contenuto della matrice. Dove c'è un uno coloreremo la cella di un colore vivo, altrimenti la lasceremo del colore dello sfondo.

Modifichiamo il programma `life.js` nel seguente modo:

- All'inizio aggiungiamo la dichiarazione della variabile `lifeA` che rappresenta la nostra matrice della vita. Aggiungiamo anche la definizione dei colori che vogliamo usare per colorare le celle (prendendoli dal tema solarized):

```
var lifeA = [];                  // <-- aggiunta al passo 2

var cellAliveColor = "#cb4b16";  // <-- aggiunta al passo 2
var cellDeadColor = "#073642";   // <-- aggiunta al passo 2
```

- Modifichiamo quindi la nostra funzione principale aggiungendo alla fine le seguenti istruzioni:

```
  // Prepara la matrice Life
  allocaMemoriaPerMatrice(lifeA);          // <-- aggiunta al passo 2

  // colora un po' di celle a casaccio
  inizializzaMatriceLifeACaso(lifeA);      // <-- aggiunta al passo 2
  coloraCelleInBaseAMatrice(lifeA);        // <-- aggiunta al passo 2
```

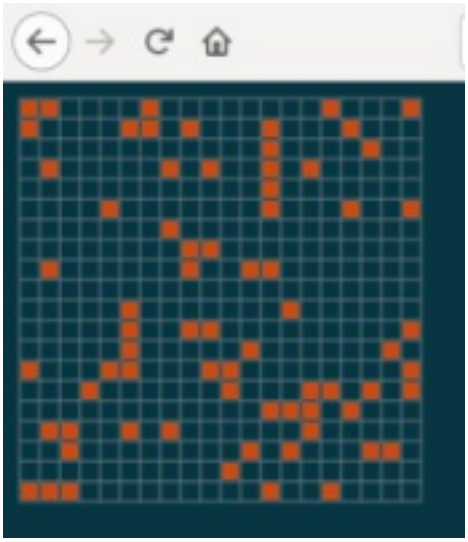
- E aggiungiamo l'implementazione delle tre funzioni `allocaMemoriaPerMatrice`, `inizializzaMatriceLifeACaso` e `coloraCelleInBaseAMatrice`:

```
// ------------------------------------------------------------------------
```

```
function allocaMemoriaPerMatrice(mtx)
{
  for (var row = 0; row < numeroRighe; row++)
  {
    mtx[row] = new Array(numeroColonne);
  }
}

// ---------------------------------------------------------------------------
function inizializzaMatriceLifeACaso(mtx)
{
  // riempie mtx[][] di 0 e 1 a caso
  //    0 significa cella morta
  //    1 significa cella viva
  for (var row = 0; row < numeroRighe; row++)
  {
    for (var col = 0; col < numeroColonne; col++)
    {
      // Random ritorna un numero da 0 a 1
      // noi vogliamo più o meno una cella viva ogni quattro
      if (Math.random() >= 0.75)
        mtx[row][col] = 1;
      else
        mtx[row][col] = 0;
    }
  }
}

// ---------------------------------------------------------------------------
function coloraCelleInBaseAMatrice(mtx)
{
  ctx.beginPath();
  for (var row = 0; row < numeroRighe; row++)
  {
    for (var col = 0; col < numeroColonne; col++)
    {
      if (mtx[row][col] != 0)
        cellAlive(row, col);
      else
        cellDead(row, col);
    }
  }
  div.appendChild(c);
}

// ---------------------------------------------------------------------------
function fillCell(i, j, color)
{
  var cellfullsize = cellsize + 1;
  var cellstartx = (j*cellfullsize)+2;
  var cellstarty = (i*cellfullsize)+2;

  ctx.fillStyle = color;
  ctx.fillRect(cellstartx, cellstarty, 9, 9);
}

function cellAlive(i,j) { fillCell(i, j, cellAliveColor); }
function cellDead(i,j) { fillCell(i, j, cellDeadColor); }
```

A questo punto proviamo il gioco: vedremo che una volta cliccato il tasto "Comincia" apparirà il campo da gioco con alcune celle colorate (più o meno una su quattro).

Ecco un esempio di una nuova griglia:



Passo 3: il gioco della vita

In questo passo vediamo il gioco della vita vero e proprio. Come abbiamo visto, si tratta di un automa cellulare in cui in ognuna delle celle del nostro tavolo da gioco ci può essere una cella viva oppure no.

La distribuzione delle cellule però evolve seguendo delle regole che dipendono dallo stato delle celle vicine. Citando Wikipedia:

- Qualsiasi cella viva con meno di due celle vive adiacenti muore, come per effetto d'isolamento;
- Qualsiasi cella viva con due o tre celle vive adiacenti sopravvive alla generazione successiva;
- Qualsiasi cella viva con più di tre celle vive adiacenti muore, come per effetto di sovrappopolazione;
- Qualsiasi cella morta con esattamente tre celle vive adiacenti diventa una cella viva, come per effetto di riproduzione.

Per calcolare un'iterazione abbiamo bisogno di una seconda matrice. Aggiungiamo quindi una variabile `lifeB[]` e modifichiamo la funzione principale nel seguente modo:

```
var lifeB = [];                  // <-- aggiunta al passo 3
...

function eseguiProgrammaLife(nrows, ncols)
{
  // salva il numero di righe e di colonne in due variabili
  // globali che saranno utilizzate nel seguito dal resto del Programma
  numeroRighe = parseInt(nrows);
  numeroColonne = parseInt(ncols);

  // nasconde i campi per la lettura delle dimensioni del campo
  MostraNascondi('N');

  // disegna il campo di gioco
  preparaCanvas();

  // Prepara la matrice Life
  allocaMemoriaPerMatrice(lifeA);         // <-- aggiunta al passo 2
  allocaMemoriaPerMatrice(lifeB);         // <-- aggiunta al passo 3
```

```
  // colora un po' di celle a casaccio
  inizializzaMatriceLifeACaso(lifeA);     // <-- aggiunta al passo 2
  coloraCelleInBaseAMatrice(lifeA);       // <-- aggiunta al passo 2

  // Codice nuovo inserito nel passo 3 ----------
  // Ciclo principale
  eseguiCicloDellaVita();
}
```

A questo punto non ci rimane altro che realizzare la funzione `eseguiCicloDellaVita()` e tutte le funzioni che ne conseguono. Vediamo il codice inserito in questo step.

- Le funzione asincrona (async) `eseguiCicloDellaVita()` calcola continuamente l'evoluzione della matrice di gioco alternando l'uso delle matrici `lifeA` e `lifeB`.
- La funzione `calcolaProssimaGenerazione()` parte da una matrice contenente lo stato corrente della griglia e popola una seconda matrice con lo stato dello step successivo applicando le regole viste in precedenze. Questa funzione, il vero cuore del nostro programma, controlla una ad una tutte le celle e in base allo stato delle celle che la circondano decide lo stato successivo.
  - se la cellula è viva, valuta se deve farla morire. Per fare questa valutazione chiama la funzione `valutaMorte()`
  - se la cellula è morta, valuta se deve farla nascere. Per fare questa valutazione chiama la funzione `valutaVita()`
- Le funzioni `valutaVita()` e `valutaMorte()` fanno uso di un paio di helper che normalizzano gli indici di riga e colonna delle celle adiacenti nel caso in cui la cella sia su un bordo o ancor peggio su un angolo. Difatti ricordiamoci che vogliamo simulare un mondo toroidale, per cui, ad esempio, una cella sul lato sinistro della matrice ha delle celle adiacenti che nella rappresentazione grafica stanno al lato opposto (sull'ultima colonna a destra).

Ecco quindi il codice inserito per il passo 3:

```
// ****************************************************************************
// DA QUI IN POI IL CODICE NUOVO DEL PASSO 3
// ----------------------------------------------------------------------------
// For promises, read:
//   https://ponyfoo.com/articles/es6-promises-in-depth
function sleep(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

// ----------------------------------------------------------------------------
async function eseguiCicloDellaVita(nrows, ncols)
{
  while (1)
  {
    await sleep(200);
    calcolaProssimaGenerazione(lifeA, lifeB);
    coloraCelleInBaseAMatrice(lifeB);
    await sleep(200);
    calcolaProssimaGenerazione(lifeB, lifeA);
    coloraCelleInBaseAMatrice(lifeA);
  }
}

// ----------------------------------------------------------------------------
function calcolaProssimaGenerazione(oldmtx, newmtx)
```

```
{
  for (var row = 0; row < numeroRighe; row++)
  {
    for (var col = 0; col < numeroColonne; col++)
    {
      if (oldmtx[row][col] == 1)
      {
        if (valutaMorte(oldmtx, row, col))
          newmtx[row][col] = 0;
        else
          newmtx[row][col] = 1;
      }
      else
      {
        if (valutaVita(oldmtx, row, col))
          newmtx[row][col] = 1;
        else
          newmtx[row][col] = 0;
      }
    }
  }
}

// ------------------------------------------------------------------------
function valutaMorte(mtx, row, col)
{
  var liveCount = 0;
  // valuta se una cella viva deve morire in base al suo intorno
  // Qualsiasi cella viva:
  //   - Se ha intorno a se meno di due celle, muore
  //   - se ha intorno a se più di tre celle vive adiacenti, muore
  for (i = -1; i <= 1; i++) {
    for (j = -1; j <= 1; j++){

      // non considera se stesso
      if ((i == 0) && (j == 0))
        continue;

      // simula la geometria di un toroide ----
      if (mtx[normalizzaRiga(row + i)][normalizzaColonna(col + j)]) {
        ++liveCount;
        if (liveCount > 3) return true;
      }
      // ------------------------------------
    }
  }

  if (liveCount < 2) return true;
  return false;
}

// ------------------------------------------------------------------------
function valutaVita(mtx, row, col)
{
  var liveCount = 0;
  // valuta se una cella vuota (morta) deve popolarsi in base al suo intorno
  // Qualsiasi cella morta:
  //   - Se ha intorno a se esattamente tre celle vive, nasce
  for (i = -1; i <= 1; i++) {
    for (j = -1; j <= 1; j++){

      // non considera se stesso
      if ((i == 0) && (j == 0))
```

```
        continue;

      // simula la geometria di un toroide ----
      if (mtx[normalizzaRiga(row + i)][normalizzaColonna(col + j)]) {
        ++liveCount;
      }
      // ------------------------------------
    }
  }
  if (liveCount == 3) return true;
  return false;
}

// Queste funzioni servono per simulare la geometria di un toroide.
// Sapreste spiegare perché?
// ------------------------------------------------------------------------------
function normalizzaRiga(row)
{
  if (row < 0)
    return numeroRighe + row;
  else
    return row % numeroRighe;
}

// ------------------------------------------------------------------------------
function normalizzaColonna(col)
{
  if (col < 0)
    return numeroColonne + col;
  else
    return col % numeroColonne;
}
```

Possibili estensioni

L'estensione ovvia è fare in modo di simulare alcuni campi di partenza che contengano delle configurazioni predefinite che ci permettano di esplorare il comportamento dell'algoritmo della vita. Per alcune di queste configurazioni si rimanda alla **pagina di Wikipedia in inglese**.

Per fare questo aggiungere dei radio button che permettano di specificare la configurazione di partenza nella pagina di setup, passare un terzo argomento alla funzione `eseguiProgrammaLife()` e modificare il file js in modo da popolare la matrice di partenza in modo diverso a seconda della scelta. .

Credits

Questo tutorial è stato realizzato dai mentor del Coderdojo di Firenze. Il tutorial e il codice sorgente del programma possono essere scaricati dall'area delle risorse del sito del Coderdojo di firenze, all'URL:

https://firenze.coderdojo.it/risorse/

Tutto il codice prodotto dal Coderdojo di Firenze per l'uso è anche liberamente disponibile nel seguente repository su Github:

https://github.com/coderdojofirenze/coderdojofi