# beTETRIS – una guida semplice di introduzione alla programmazione spicciola

**by Alessandro Gaggia**

Ciao tutti! Questa guida è stata pensata per introdurre alcuni semplici concetti di programmazione validi per moltissimi linguaggi oltre che permettere ai meno esperti di mettere le mani sul codice fin da subito e creare il loro primo gioco da zero, partendo da un classico anni 80: il TETRIS.

Perché TETRIS: questo gioco fa uso di meccaniche incredibilmente semplici, ciò ci permette di avere con poco sforzo un gioco completo e divertente in poco tempo; inoltre il funzionamento di TETRIS prevede un uso intensivo di elementi quali gli **Array** (Vettori) che sono utilizzati tantissimo in programmazione sapere come navigarli è una skill fondamentale. Inoltre vedremo l’utilizzo di **variabili**, **costanti**, **colori**, schermo, **gestione della tastiera**, **funzioni**, **oggetti** e molto altro ancora; insomma nonostante la sua semplicità è un gioco ricco di elementi interessanti e utili al nostro approccio alla programmazione.

Di questo gioco vedremo due varianti la prima è una prototipazione rapida che ci permetterà di giocare a TETRIS sin da poche righe di codice, poi andremo a rifinire il tutto con una versione 2.0 che introdurrà alcuni miglioramenti grafici e interattivi.

La scelta del linguaggio: al giorno d’oggi esistono tanti linguaggi di programmazione quanto i granelli di sabbia di una spiaggia, ognuno di essi ha pregi e difetti, ma la REGOLA di base che va sempre seguita è che ogni linguaggio ha un suo subset di casistiche in cui può esprimersi al meglio e un buon programmatore SA che conoscere più di un linguaggio aiuta tantissimo a destreggiarsi tra le varie richieste di lavoro e problemi da affrontare.

Nel nostro esempio andremo ad utilizzare un linguaggio di **scripting** (non di programmazione per sé e vedremo la differenza) usatissimo e che ha generato una notevole schiera di fan (e di haters) JAVASCRIPT.

Perché JAVASCRIPT? Semplicemente perché ad oggi è sicuramente il linguaggio più immediato per un principiante, dato che ognuno di noi avendo un PC di qualsiasi marca e sistema operativo, è già fornito degli strumenti per programmare efficacemente in JAVASCRIPT: un editor di testo e un browser.

Oltre a ciò JAVASCRIPT non necessita di IDE (programmi avanzati per la scrittura di codice che di solito gestiscono anche quello che chiamiamo **progetto di lavoro**), ne di essere “**compilato**” o “**buildato**” per funzionare e questo ci favorisce nel vedere subito il frutto del nostro lavoro man mano che programmiamo.

Prima di iniziare però vorrei fare una breve introduzione a quelli che secondo me sono i temi fondamentali che chiunque si avvicini alla programmazione deve quantomeno conoscere; partiamo da che cosa vuol dire programmare in sé per sé: programmare per noi significa inventare una soluzione ad un problema per computer; sia esso un programma che effettua somme, uno che permette di mostrare un quadrato a schermo oppure un complesso gioco per PC, la base è sempre la stessa: qualcuno ci chiede “vorrei questa cosa se faccio così” e noi la programmiamo in un linguaggio che ci pare opportuno.

Quindi che cosa ci serve per ”**programmare**”? Innanzitutto dobbiamo immaginare i passi che andremo a svolgere per risolvere un determinato problema e poi pensiamo a come convertirli in codice. Quindi programmare non è diverso da cucinare, disegnare, suonare uno strumento o addestrare un cane a compiere difficili esercizi a comando. Si tratta solo di conoscere o quantomeno preventivare gli step necessari a raggiungere un obiettivo.

Perciò quando si vuole programmare la prima fase è la progettazione cartacea del problema; prendiamo una risma di fogli di carta e cominciamo a descrivere gli elementi che andremo ad implementare e, solo dopo, cominciamo a scrivere.

Come organizziamo allora il nostro codice? Di solito un buon programmatore è ordinato e organizza il proprio codice in quelli che vengono definiti **PROGETTI**; nel nostro caso un progetto è una semplice cartella nel computer in cui andremo a scrivere i nostri file, ma per progetti complessi di solito si fa riferimento ad un set di cartelle e file con nomenclature specifiche che programmi appositi riconoscono e organizzano per noi (i famosi IDE come **IntelliJ** o **Rubymine** che sentite spesso citati dai nostri programmatori in ufficio).

Abbiamo parlato di suddividere il nostro obiettivo in una serie di step: questo è un buon metodo perché possiamo idealmente associare i vari step a **FUNZIONI** (anche chiamate **METODI**), ovverosia a “**comandi complessi**” che racchiudono diverse operazioni.

Utilizzare una serie di funzioni in un ordine preciso o in risposta ad **EVENTI** ci assicura di raggiungere il nostro obiettivo e di avere un programma funzionante.

# Terminologia Spicciola

Parliamo un pochino di terminologia:

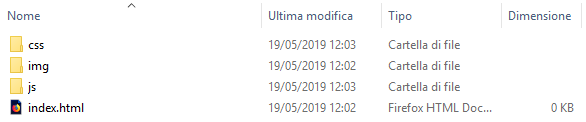
* **VARIABILE**: una variabile è un elemento finito che racchiude un valore. Essa può essere un semplice numero o una stringa di testo, queste vengono chiamate variabili semplici o di TIPO perché fanno riferimento ad elementi universalmente riconosciuti come fondamentali alla programmazione, infatti se ci pensiamo ogni programma in fondo è un insieme di bit cioè di 1 e 0, numeri! Altre variabili possono essere COMPLESSE cioè fare riferimento ad un costrutto creato da noi ospitando VETTORI, FUNZIONI od OGGETTI.
* **COSTANTI**: una costante è una variabile che non “varierà” più: di solito un numero speciale o una stringa di testo predefinita; il vantaggio di definirla come costante? Per noi “umani” è che sappiamo che non va più modificata una volta creata; per il computer significa ottimizzarne il suo funzionamento all’interno del programma facendolo girare “meglio”.
* **FUNZIONE** (O METODO): una funzione è un comando complesso che racchiude più righe di codice e permette di eseguire operazioni complesse in un momento specifico del programma in quanto può essere INVOCATA (o CHIAMATA) a nostra discrezione. E’ come immaginare di aver insegnato ad un cane che “RIPORTA” significa: cerca la pallina, prendila e portala indietro; una serie di operazioni diverse ma con un obiettivo comune che il cane (il nostro programma) “sa” e che noi INVOCHIAMO quando rivogliamo la pallina.
* **ARRAY** O VETTORI: sono liste di elementi caratterizzati da una posizione all’interno della lista, si usano spessissimo perché sono come contenitori organizzati di informazioni che verranno richiamate in seguito in modo intelligente e definito. Se pensiamo di mettere diversi VETTORI dentro un altro VETTORE di fatto stiamo immaginando una MATRICE righe e colonne.
* **STRINGHE**: righe di testo che serviranno all’interno del programma.
* **OGGETTI**: nella programmazione detta ad oggetti oltre a poter organizzare il codice all’interno di FUNZIONI, possiamo organizzarlo ulteriormente a livello di OGGETTI, che sono elementi complessi dotati di tante funzioni e VARIABILI peculiari. Ad essi associamo un nome così da poterli identificare all’interno del nostro programma e una volta creati essi diventano delle VARIABILI complesse così da poter essere manipolati.
* **CLASSE**: una classe è un altro modo per definire un oggetto in determinati linguaggi. Ha alcune proprietà particolari come L’ereditarietà ovvero gli oggetti possono dipendere da altri oggetti padri e senza scrivere di nuovo lo stesso codice possono utilizzarne i metodi e le variabili.
* **PUBBLICO E PRIVATO**: rappresentano un modo per dirci se una variabile o una classe può essere vista in tutti i punti del programma oppure no, questo permette al programmatore di essere sicuro che una variabile non interferisca con altre sparse in giro per le 1000000 righe di codice che ha scritto per il giorno dopo :D.
* **CICLO FOR E FOREACH**: il ciclo è una operazione ripetuta un numero preciso di volte e viene utilizzato spessissimo per effettuare somme, creazione di oggetti ripetuti come liste, o ispezionare gli elementi di un array o di una matrice. Ovviamente se pensiamo ad un qualcosa che viene mostrato a schero il ciclo serve ad esempio per dipingere tutti i pixel dello schermo!
* **WHILE DO**: il while è un ciclo di cui non è detto si conosca a priori la file infatti di solito nei vari linugaggi il ciclo while possiede una condizione di uscita! Ovvero: se succede questo allora smetti.
* **IF ELSE**: chiudiamo questa prima carrellata di nomi con le condizioni if else: come si può facilmente intuire IF ed ELSE permettono di eseguire del codice se e solo se le loro condizioni sono verficate: si possono leggere come se succede qualcosa (IF) fai questo altrimenti (ELSE) fai quest’altro.

**Ora direi che abbiamo studiato abbastanza per cominciare: è il momento di partire con la prototipazione degli elementi e del progetto!**

## Costruzione della struttura di progetto!

Cominciamo col preparare la nostra cartella di progetto: scegliete un punto comodo del vostro computer ad esempio il Desktop (ma da buon programmatore ordinato perché non crearsi una cartella di progetti sul vostro disco e **lì mettere** la nostra cartella che chiameremo **beTetris**?)

1. Creare una cartella beTetris e all’interno predisporla come segue:



* Creiamo 3 cartelle **css**, **img** e **js**
* Creiamo un file vuoto **index.html**

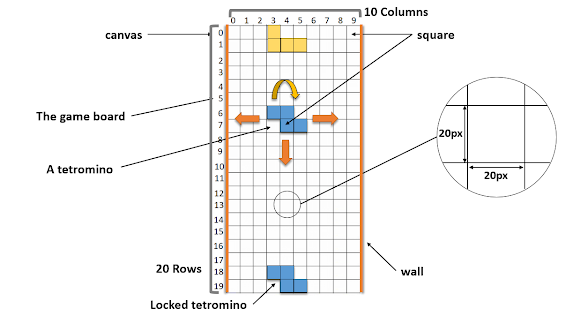
1. Creiamo un file **tetris.css** nella cartella css e uno chiamato **tetris.js** nella cartella js
2. Ora predisponiamo il nostro ambiente di progetto: apriamo il nostro editor di testo e un browser a scelta (consiglio rispettivamente ATOM <https://atom.io/> e Chrome o Firefox).

**Siamo pronti a cominciare la progettazione del gioco!**

## Progettiamo Tetris!

Come funziona Tetris? Abbiamo un’area di gioco e diversi tasselli che vengono calati dall’alto in maniera casuale; i tasselli cominciano a scendere e il giocatore può spostarli e ruotarli. Quando i tasselli raggiungono il fondo si bloccano e quando una riga completa viene creata si distrugge e si aggiungono dei punti al giocatore. Quando non abbiamo più spazio per posizionare l’ultimo tassello abbiamo perso la partita!

Abbiamo dunque individuato alcuni elementi fondamentali:



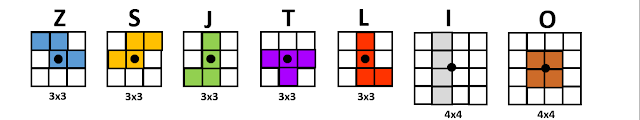
* Il campo di gioco: è un rettangolo di **20** **righe** per **10** **colonne**.
* Un tassello (o tetromino): è un oggetto composto da piccoli rettangoli ed un colore.
* Dobbiamo muoverci in 3 direzioni: SX, DX e GIU’.
* In basso il tassello si blocca.
* Quando non possiamo più posizionare gli elementi il gioco termina
* Quando una riga composta da quadratini è piena si elimina e si prendono punti

Quello che si può notare fin da subito è che sia i tasselli che il campo di gioco sono formati da quadrati di dimensione arbitraria (nel nostro esempio 20x20px perché stiamo lavorando a schermo e lo schermo utilizza proprio il pixel come unità di misura della sua area di presentazione delle immagini)

Possiamo quindi pensare per descrivere le nostre operazioni l’unità di misura da programmare sarà uno **SQUARE** (un quadrato di dimensioni 20x20).

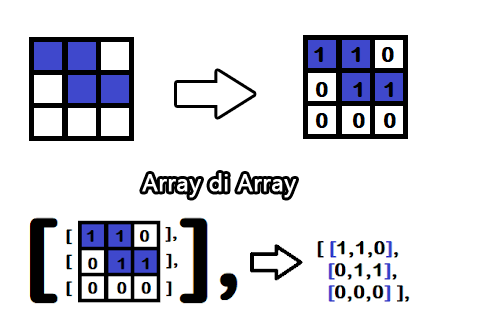
Inoltre il nostro campo di gioco (che verrà presentato su di un **canvas,** un elemento Html che permette di disegnare oggetti sulla pagina Html) può essere rappresentato come una matrice 20x10 (o 10x20). Se ci ricordiamo della definizione di Array data pocanzi, significa creare un Array di Array!

**Tetromini:** sono 7 pezzi differenti con cui giochiamo: **Z, T, L, O, I, J** e **S**. Questi pezzi possono muoversi e ruotare. Quando un Tetromino raggiunge il fondo ne creeremo un altro in cima!

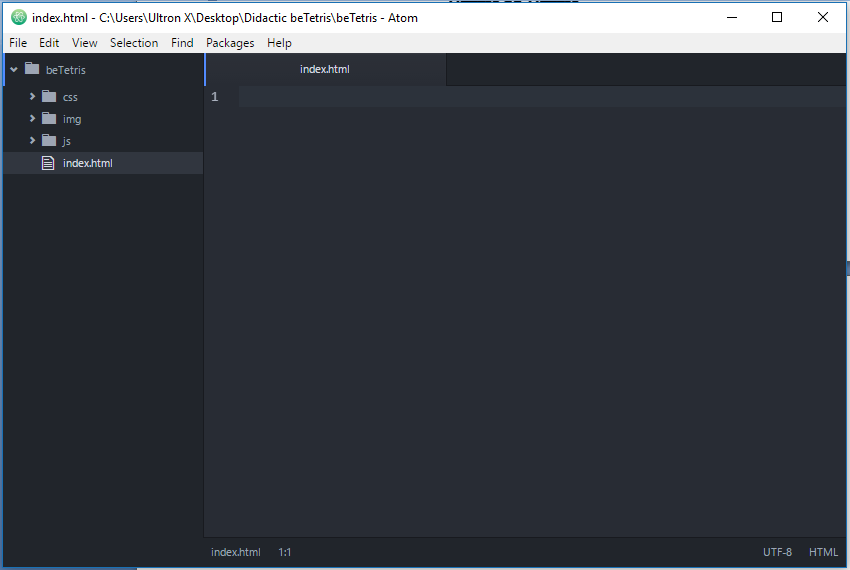


Il puntino nero nell’immagine indica il punto di rotazione.

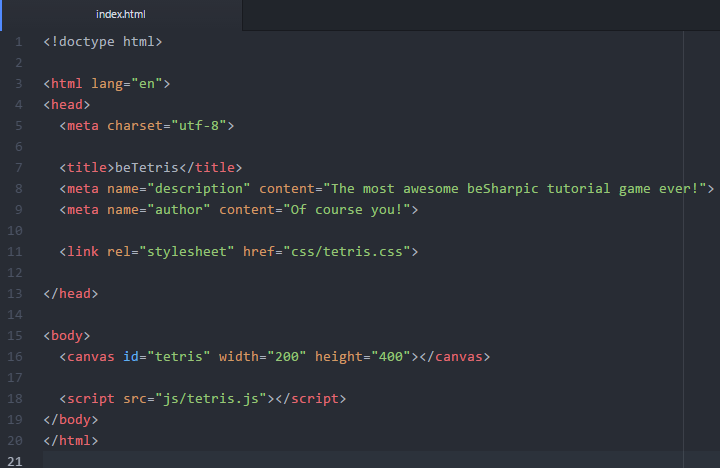
Se immaginiamo di rappresentare ogni quadratino dello schermo di gioco come un elemento con valore 0 (vuoto) e 1 (pieno) allora:



Ok cominciamo a programmare sul serio! Cliccate col tasto DX sulla vostra cartella **beTetris** e scegliete l’opzione **apri con Atom**. Oppure aprite Atom e dal menù “file” scegliete “open folder”, vi ritroverete così ad avere il progetto intero aperto in Atom per vostra comodità in questo modo:



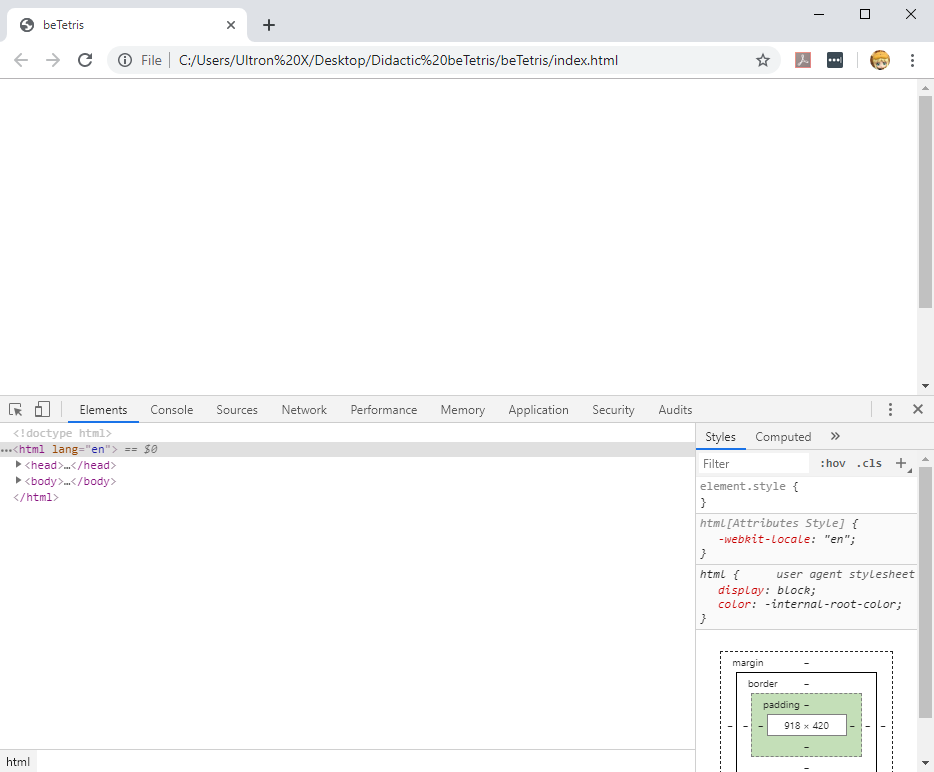
Cominciamo a preparare il nostro Canvas di gioco e ad importare tutti i file necessari: questo procedimento è valido sempre quindi lo riutilizzeremo più volte man mano che costruiremo il nostro codice! (il codice è già presente nella cartella di progetto allegato al file README)



Cosa stiamo facendo qui? Abbiamo creato una pagina html con alcuni campi nella sezione **<head**>. La sezione head tipicamente contiene il tag **title**  che permette di mostrare il titolo sulla tab del browser alcuni metatag che vengono letti ed analizzati dai motori di ricerca; i fogli di stile (stylesheets) o **css**  che permettono di impreziosire la grafica della pagina con alcune opzioni di stile che vedremo.

La sezione **<body>** contiene il nostro oggetto **canvas** e soprattutto uno dei nostri script (ne aggiungeremo altri allo stesso modo più avanti). Questa struttura base è valida per molte pagine web moderne quindi in genere è così che andrete a strutturare anche le pagine di vostri siti, etc (se siete interessati fate qualche ricerca su HTML5 Boilerplates).

Se salviamo il documento html e lo apriamo in Chrome con tasto DX “apri con -> Chrome” avremo questa situazione: una pagina bianca :D. Ciò che mi interessa però è mostravi uno strumento indispensabile al programmatore Web: la console di debug di Chrome! Cliccate col tasto DX sulla pagina e selezionate “inspect” vi ritroverete davanti:



Mediante questa console potrete tenere d’occhio lo stato di sviluppo del vostro gioco, troverete eventuali errori, analizzerete l’html della pagina e molto altro.

Ora cominciamo a creare i nostri elementi di gioco: il campo e i tetromini.

Andiamo in Atom e aggiungiamo il seguente file **tetrominoes.js** nella cartella **js** e all’interno i seguenti elementi. (ricordatevi inoltre di aggiungere lo script SOPRA tetris.js nel file index.html)

**const** I = [

[

[0, 0, 0, 0],

[1, 1, 1, 1],

[0, 0, 0, 0],

[0, 0, 0, 0],

],

[

[0, 0, 1, 0],

[0, 0, 1, 0],

[0, 0, 1, 0],

[0, 0, 1, 0],

],

[

[0, 0, 0, 0],

[0, 0, 0, 0],

[1, 1, 1, 1],

[0, 0, 0, 0],

],

[

[0, 1, 0, 0],

[0, 1, 0, 0],

[0, 1, 0, 0],

[0, 1, 0, 0],

]

];

Possiamo fare riferimento al file completo dell’esempio proposto per aggiungere tutti i tetromini in una volta, ma provate da soli ad aggiungere gli altri seguendo questo schema e solo dopo confrontate con l’esempio completo.

Ora che abbiamo i pezzi possiamo provare a costruire il nostro gioco. Torniamo al file **tetris.js** e aggiungiamo per cominciare alcune costanti e variabili principali del gioco:

// We first need to define two constants for our game canvas:

// The canvas itself which is saved to a constant and the 2d

// context in which we will be drawing the elements of our game

// Using document.getElementById("id\_value") we can get every element

// in the html page that has a particular id, see the tetris object in

// index.html as a reference. To do this the script must be included in

// that specific pageas we have done with index.html

**const** cvs = document.getElementById("tetris");

**const** ctx = cvs.getContext("2d");

// We also save the score graphic element

**const** scoreElement = document.getElementById("score");

// The number of rows and the number of columns. Remember when

// we talk about constants? Here we are defining some of them.

// They can be changed ONLY before starting the game and ONLY

// by the programmer. The idea behind these is to customize your

// game somehow!

**const** ROW = 20;

**const** COL = COLUMN = 10;

// Since the board and the pieces are both made of squares, we will

// use a constant called SQ (for square) = 20, as a unit.

// We can make also what I call a double association of a variable

// as we can assign the same value to two element.

**const** SQ = squareSize = 20;

// A VACANT (empty) square has a white (#FFF) color

**const** VACANT = "#FFF";

// The score value is a let because it will change during the game

**let** score = 0;

// Now we define the board array. 'let' is a way to say a variable

// in JS Ecmascript 6 (another way for old JS is 'var').

// By assigning a [] to a variable we are defining an empty array

**let** board = [ ];

Ora che abbiamo definito alcune variabili principali è ora di creare il nostro schema vero e proprio, riempiamo insieme la board di tasselli bianchi!

// We create the board as a Matrix: as we said before the idea is making

// an Array of Arrays.

// let's create the rows.

// This is how we define an increasing for loop:

// for(let index = 0; index < something; index++) {}

// This is how we define a decreasing for loop:

// for(let index = something; index > 0; index--) {}

// Sooo we start from 0 and we make all the rows...

**for** ( **let** r = 0; r < ROW; r++) {

// every single row is an empty array. note that array[index] is

// the way to access or set its value in that position

board[r] = [ ];

// let's create the columns

**for**( **let** c = 0; c < COL; c++){

// This is how we access a multi dimensional array: array[first\_index][second\_index].

// Like we do with real matrices

board[r][c] = VACANT;

// when we first draw the board all the square are empty, so every square has the value "#FFF".

}

}

Ora definiamo come disegnare un semplice quadrato all’interno della board utilizzando un’altra funzione:

// As you can see the code can be operated as stand alone by simply writing it in the page but a more elegant

// way is to put it inside functions. This is also important if we want to do the same thing more than one time!

// draw a square: this is an helper function; whenever we need to tidy our code we use a function

// that group some common operations. In this case the function accepts 2 coordinates and a color

// as parameters and use special canvas functions to draw a square on the canvas

**function** drawSquare(x,y,color) {

// We set the color for drawing

ctx.fillStyle = color;

// we draw a filled rectangle with this method note that the position

// of the element in the grid is multiplied by our unit the SQUARE

ctx.fillRect(x\*SQ, y\*SQ, SQ, SQ); // x, y, width, height

// The stroke color for the borders

ctx.strokeStyle = "BLACK"; // this is what is called a "magic string", or a string that remains in a

// part of the code not specifically controlled by the programmer and thus

// is prone to possible errors and can't in general be reused. A solution is

// to put that value in a constant at the top of our code as the rest of our

// constants and set the constant here instead, try it!

// Then we stroke the same square element insted of filling it

// the result is a colored square with a black border

ctx.strokeRect(x\*SQ, y\*SQ, SQ, SQ);

}

Questo ci permetterà in seguito di generare facilmente quadrati colorati per rappresentare il nostro Tetromino!

Di seguito aggiungiamo un’altra funzione che permette di disegnare tutto lo stato corrente della board di gioco usando la funzione di prima come helper:

// draw the board: to draw the board to the screen we use a function:

// as we said before a function is a set of operation that we can call

// at a specific time to do something specific; in this case for every

// element of the game board: we draw a square taking the coordinate and

// the board value into account.

**function** drawBoard() {

// As we have seen before we cycle through the entire board with 2 for loops

**for**( r = 0; r <ROW; r++){

**for**(c = 0; c < COL; c++){

// Then we draw a square by using another function.

// And yes of course one function can call another one.

// This approach can help use organize and tidy up our code

drawSquare(c,r,board[r][c]);

}

}

}

Una volta definita la funzione, chiamiamola per preparare la board all’inizio del gioco!

// After the declaration we can draw the board. This is how we call a function.

// functionName(); if a function requires parameter at launch you will pass those

// parameters inside the parenthesis.

drawBoard();

Ora definiamo un **array** di pezzi del gioco facendo riferimento alle costanti create nel file **tetrominoes.js**

// Sooo the board is drawn with all basic white squares because we defined our

// board as all white elements in the beginning.

// Let's define the pieces and their colors. We prepare a constant array that includes

// all of our pieces with a specific color. Note that we are calling the tetrominous from the other file

// tetrominoes.js which need to be called in the index.html page as well. Also plese remember the order in which

// you call the js file to be sure that the variables, object and functions are all called AFTER they are defined

// in your code. Otherwise you'll face errors in your browser.

**const** PIECES = [

[Z,"red"],

[S,"green"],

[T,"yellow"],

[O,"blue"],

[L,"purple"],

[I,"cyan"],

[J,"orange"]

];

Ora la parte più lunga: andremo a definire una **classe** pezzo (Piece) che definirà i comportamenti e le proprietà del nostro Tetromino. Ricordate quanto detto in precedenza? E’ possibile raggruppare metodi (funzioni) e variabili all’interno di oggetti complessi che a loro volta possono essere resi variabili per altre funzioni . La parte che andrete a leggere di seguito e che potete incollare nel vostro file tetris.js è molto lunga ma estremamente ben commentata, consiglio di leggere attentamente ogni commento legato alle righe di codice per capire cosa stiamo costruendo! La cosa migliore è incollare il blocco in Atom e poi leggere i commenti e le funzioni da lì!

// Ok in the function randomPiece we will call new Piece(....) BUT, what is new Piece, we have

// to define it. It is a complex element of our game and it is a collection of properties and functions. Soooo

// as we have learned from the README file, a structure with some properties and methods is an OBJECT or in our case

// a class which is more elegant. Let's build it!

// The class Piece

**class** Piece {

// Constructor is as the name implies the master method of the class and is called when we do new Piece(piece, color)

// remember the random piece generator function? When we add elements to the constructor we use the keyword "this"

// This special word means that we are assigning something to the object itself and, inside this object code, we can always

// retrieve all the elements and properties saved in this! Remember when we talk in the README about variable scope? (public or private)

// this.something is another way of giving a scope to a variable which is called "instance variable" or a variable

// that is accessible inside the whole object

constructor(tetromino, color) {

**this**.tetromino = tetromino;

**this**.color = color;

**this**.tetrominoN = 0; // we start from the first pattern, the first type of rotation

// We access the piece by extracting it from a what? Surprise an array, what else?

**this**.activeTetromino = **this**.tetromino[**this**.tetrominoN];

// We need to control the start position the pieces by setting the globla coordinate of the piece itself.

// If we imagine that our start is at 0,0 (left, top) these coordinates means to go up of 2 and right of 3

**this**.x = 3;

**this**.y = -2;

}

// The fill function

fill(color) {

// We are using the same length for rows and columns just because our matrix are squared so nrows = ncolumns

**for**(r = 0; r < **this**.activeTetromino.**length**; r++){

**for**(c = 0; c < **this**.activeTetromino.**length**; c++){

// we draw only occupied squares: remember if works with true,

// but in programs 1 value is also treated as true and 0 as false

**if**(**this**.activeTetromino[r][c]){

// Remember drawSquare? We are using it now here! So we are drawing a colored

// square for every one inside the tetromino

drawSquare(**this**.x + c, **this**.y + r, color);

}

}

}

}

// Now we define two method that are extensions of the previous one, which is de facto an helper of these two

// draw a piece to the board

draw() {

**this**.fill(**this**.color);

}

// undraw a piece from the board

unDraw() {

**this**.fill(VACANT);

}

// collision function

collision(x, y, piece) {

**for**(**let** r = 0; r < piece.**length**; r++){

**for**(**let** c = 0; c < piece.**length**; c++){

// if the square is empty, we skip it

**if**(!piece[r][c]) {

**continue**; // note that continue means skip this step of the for loop and go on!

}

// coordinates of the piece after movement

**let** newX = **this**.x + c + x;

**let** newY = **this**.y + r + y;

// conditions

**if**(newX < 0 || newX >= COL || newY >= ROW) {

// over the corners? so collision! Return true and stop the method!

**return** true;

}

// skip newY < 0; board[-1] will crush our game: because an array always start from 0!!!!!

// negative values as index are called outOfBoundExceptions!

**if**(newY < 0){

**continue**;

}

// check if there is a locked piece already in the same place

**if**( board[newY][newX] != VACANT) {

// collision stop and return!

**return** true;

}

}

}

// Free to go!

**return** false;

}

// move Down the piece

moveDown() {

// If the piece is free move down otherwise lock it

**if**(!**this**.collision(0, 1, **this**.activeTetromino)) {

// We undraw before moving to avoid artifact on screen because

// basically we don't have "layers" we draw everything on the board

**this**.unDraw();

**this**.y++;

**this**.draw();

} **else** {

// we lock the piece and generate a new one

**this**.lock();

p = randomPiece();

}

}

// move Right the piece: see how after we defined many functions creating a

// new one is just a matter to put them togheter to make the action. That is why we need to always

// put our code inside functions, and as I said before, creating a program is just a matter to divide

// a bigger problem in smaller ones and make a function for every piece of it!

moveRight() {

**if**(!**this**.collision(1, 0, **this**.activeTetromino)) {

**this**.unDraw();

**this**.x++;

**this**.draw();

}

}

// move Left the piece: same as right but we change the x with a minus

// Note that the collision algorithm basically try to see if the next

// position is empty or not before actually moving to it!

moveLeft() {

**if**(!**this**.collision(-1,0,**this**.activeTetromino)){

**this**.unDraw();

**this**.x--;

**this**.draw();

}

}

// rotate the piece: we get the next tetromino in the sequence but with a nice touch:

// if we go past the length of the array we start again from 0 with a single line of code.

rotate() {

// Let's see how: (this.tetrominoN + 1) % this.tetromino.length

// We access the next piece n + 1 and we check for the remnant of the division so let's try it:

// 0 --> 0 + 1 = 1 --> 1 % 4 = 1

// 1 --> 1 + 1 = 2 --> 2 % 4 = 2

// 2 --> 2 + 1 = 3 --> 3 % 4 = 3

// 3 --> 3 + 1 = 4 --> 4 % 4 = 0

// Nice, isn't it? The beauty of code

**let** nextPattern = **this**.tetromino[(**this**.tetrominoN + 1) % **this**.tetromino.**length**];

**let** kick = 0;

// When we rotate we need to check for walls, position is the same but different rotation

**if**(**this**.collision( 0, 0, nextPattern)) {

**if**(**this**.x > COL/2){

// it's the right wall

kick = -1; // we need to move the piece to the left

} **else** {

// it's the left wall

kick = 1; // we need to move the piece to the right

}

}

// check the new collision with the offset value

**if**(!**this**.collision( kick, 0, nextPattern)) {

// We are ok soooo undraw

**this**.unDraw();

// Shift a little

**this**.x += kick;

// Get the new one

**this**.tetrominoN = (**this**.tetrominoN + 1) % **this**.tetromino.**length**; // (0+1) % 4 => 1

// Set it

**this**.activeTetromino = **this**.tetromino[**this**.tetrominoN];

// Draw it!

**this**.draw();

}

}

// Lock the piece in position and make some calculation for the game

lock() {

// For the active tetromino we check all its colored squares

**for**(**let** r = 0; r < **this**.activeTetromino.**length**; r++){

**for**(**let** c = 0; c < **this**.activeTetromino.**length**; c++){

// we skip the vacant squares, we use continue to be faster,

// because less operation will be done by the browser

**if**( !**this**.activeTetromino[r][c] ) {

**continue**; // this is a way to ask for the next for loop step without doing anything else

}

// pieces to lock on top = game over: the condition is saying

// if the piece is over 0 so the top most value of the game screen

**if**(**this**.y + r < 0){

**alert**("Game Over");

// stop request animation frame

gameOver = true;

// break is used to force an exit from a for loop! because we don't need any other analysis!

**break**;

}

// if there is no problem we lock the piece: we set the color of the board to NOT white!

// Note how we are accessing the board matrix

board[**this**.y + r][**this**.x + c] = **this**.color;

}

}

// Remove full rows and give points: check the entire board by going from 0 to ROW

**for**(**let** r = 0; r < ROW; r++){

**let** isRowFull = true;

// and all the columns for that row

**for**(**let** c = 0; c < COL; c++) {

// isRowFull starts from true because this is an invalidating condition:

// we set the value of the variable to itself AND (&&) the color of that board position is NOT white?

// Sooo we will have something like this: imagine that Y, B, R are colors and 0 is white

// R R B 0 R R B B B Y Y B R Y Y 0, when we check all the columns here the value of the isRowFull will be

// isRowFull = true && (R != W) = true && true = true

// isRowFull = true && (R != W) = true && true = true

// isRowFull = true && (B != W) = true && true = true

// isRowFull = true && (W != W) = true && false = false

// after that the value will always remain false no matter what because:

// True AND True = True

// True AND False = False

// False AND True = False

// False AND False = False

// So the only way to have true in the end is to have a full row!

isRowFull = isRowFull && (board[r][c] != VACANT);

}

// If the row is full

**if**(isRowFull) {

// We move down all the rows above it

**for**(**let** y = r; y > 1; y--){

**for**(**let** c = 0; c < COL; c++){

board[y][c] = board[y-1][c]; // note y - 1!!!

}

}

// the top row board[0][..] has no row above it, because we move everything down by 1!

**for**(**let** c = 0; c < COL; c++){

board[0][c] = VACANT;

}

// increment the score

score += 10;

}

}

// update the board

drawBoard();

// update the score: innerHTML is a way of writing inside a tag element in the page

scoreElement.innerHTML = score;

}

}

Siamo quasi alla fine del nostro tutorial su come creare un semplice Tetris via Html 5 e javascript, ma mancano alcuni semplici elementi ancora: un generatore random di pezzi, i controlli, il ciclo di gioco! E allora via, avanti Savoia!

Generatore di pezzi random:

// We have prepared all of our pieces prototypes, let's make a function that calls for a

// random piece form the list; this piece will be the current controllable piece

// Generate random piece function

**function** randomPiece() {

// r will be our random piece chosen from our array of piece over this function

// we call a mathematical function Math.random() to produce a number between 0 and 1 but we

// want to select a number between 0 and 6 because the number of pieces are 7 and arrays are 0

// indexed which means that they start from position 0. So let's add a little operation that helps achieve this result

// we all know from math that to get a Random Arbitrary Value between a min and a max we do

// Math.random() \* (max - min) + min; BUT in this case the minimum is 0 so random \* (max - 0) + 0 or simply:

**let** r = Math.**floor**(Math.**random**() \* PIECES.**length**) // from 0 to 6

// now we can return a piece. To do so a function can use the special word: return which can return every object

// we want, in this case a piece. In fact we call new Piece to define a new Object of type Piece passing the

// tetrominoe definition and its color; in fact if we check our array of arrays with the first index, the random one we

// select a row for example, imagine that r = 3 so we get this row [O,"blue"], then with 0 as the second index we get O piece and

// with 1 as the index we get "blue"

**return** **new** Piece(PIECES[r][0], PIECES[r][1]);

}

// Get the current random piece for the start of the game. Note that this

// way of putting code in a file is not particularly elegant as we are using random

// pieces of code outside functions. Nonetheless we are creating the pieces of our game as we go so

// this approach is more "natural" for a beginner

**let** p = randomPiece();

Controllo della tastiera:

// CONTROL the piece: this is an interesting part. We are calling a JS specific function

// which is document.addEventListener(event, function\_what\_to\_do\_when\_occurs). We are saying

// that when a key on the keyboard is pressed call the Controls function

document.addEventListener("keydown", controls);

// This is the function that defines our movements as per specification every function

// passed to addEventListener accept an event parameter. When a function is defined by other programmers like this one and when

// you code it you have to abide to a specific method signature (a list of parameters specific to that function) then you are probably

// facing a callback! A callback is a function that is called on specific moment by the program itself and not directly by you!

// A good example is if you have a dog or a cat (our program) and he/she suddenly start calling for your attention not by your

// specific command. He/she is doing it in response to something internal to Him/Herself; this is the same

**function** controls(event) {

// Here we are making adjustments to the piece position and rotation based on which key the user pressed

// We also change the value of dropStart because we have change the board and we need to purge a refresh,

// thus changing this value. We don't need to call for the drop method below because it is called in a loop

// so the value of dropStart is always recovered there

**if**(event.keyCode == 37) {

p.moveLeft();

dropStart = Date.now();

// Note the use of else if construct that basically say us to make a chain of conditions: if is not this one

// is this other one and so on

} **else** **if**(event.keyCode == 38){

p.rotate();

dropStart = Date.now();

} **else** **if**(event.keyCode == 39){

p.moveRight();

dropStart = Date.now();

} **else** **if**(event.keyCode == 40){

p.moveDown();

}

}

E per finire il ciclo di gioco:

// drop the piece every 1 sec

// Get a time event as Now, usually this means that the computer

// will get the time as the interpreter read this specific line.

// Is used as a way to get the current datetime on a machine. Imagine

// that you want to make a clock in JS, this could simplify a lot your code,

// just get this value every second and display it!

**let** dropStart = Date.now();

// We are not a loser now, put this variable to false

**let** gameOver = false;

// This function create a basic game loop to run our game: we ask for the "requestAnimationFrame"

// which is a specific function of the browser and we pass our drop function as a parameter to it.

// It tells the browser to let the program draw something on screen using the browser capabilities

// to communicate with the video card of the computer, assuring that we get a constant framerate and

// other useful improvement to the quality of the result.

**function** drop() {

// Get another instant time

**let** now = Date.now();

// Check how many milliseconds are passed between our start time and now

**let** delta = now - dropStart;

// If 1 sec (1000 millisec) is passed

**if**(delta > 1000) {

// Move the piece down a row in the screen

p.moveDown();

// Get a new current time for the next time difference, so we can define an "infinite loop" of checks

// You can see that we are overriding the value of dropStart so next time this variable will have this value

dropStart = Date.now();

}

// If we are NOT in game over (! is a simple way to turn false to true and true to false)

// We are doing this because "if" only works with true values so we are negating the gameOver false to make it true for the if

// condition

**if**(!gameOver) {

// request a game frame

requestAnimationFrame(drop);

}

}

// Start the game! Calling drop function we basically start the game loop: by

// doing this all the functions will be called at the right moment thus giving

// life to the program itself

drop();

Il codice del gioco è complete; come si può capire tolti gli estensivi commenti la logica di gioco è veramente poca, non di meno interessante sotto diversi spunti! Ora per completare il tutto, aggiungiamo un file **tetris.css** nella cartella **css** se non ancora presente e al suo interno aggiungiamo due regole di stile per allineare il campo di gioco allo schermo:

#score{

display: inline-block;

}

**div** {

font-size: 25px;

font-weight: bold;

font-family: monospace;

text-align: center;

}

**canvas** {

display: block;

margin:0 auto;

}

E verifichiamo di aver aggiunto anche l’oggetto score alla nostra pagina html, andiamo in index.html e aggiungiamo:

<**div**>

Score : <**div** id="score">0</**div**>

</**div**>

Sotto il tag canvas. Salviamo tutto e proviamo il nostro gioco su Chrome, come sempre tasto DX su index.html “apri con” 🡪 “Chrome”. Oppure se avevate lasciato aperta la pagina precedente refreshate con F5 o command+R o meglio ancora Shift+Control+R per Windows o Shift+Command+R per Mac!

Buon divertimento!