Week 1 : Introduction

This week we will look into a brief history of JavaScript games, and present the basic principles we will find in nearly all video game that uses real-time animation and interaction.

We will see a brief introduction on how to perform real time animation with the HTML5 canvas and manage different type of user inputs.

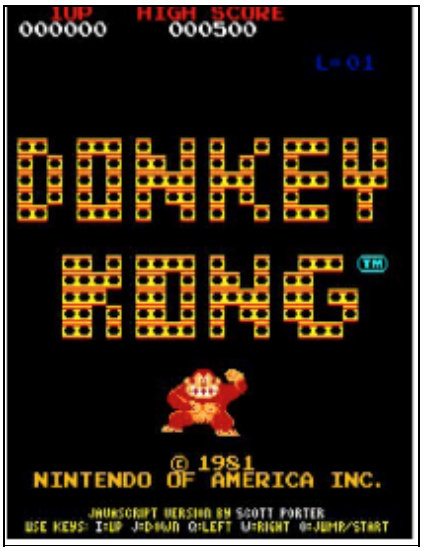
Site:	Classrooms - Online training for Web developers
Course:	HTML5 Games - November 2014
Book:	Week 1 : Introduction
Printed by:	Michel Buffa
Date:	Wednesday, 7 January 2015, 8:28 PM

Table of contents

1 History of JavaScript Games

History of JavaScript Games

People often think that games in the Internet browsers without any plugins are relatively new phenomenon of web development. It's not true.



It is not the first Web revolution in our history. Just after the Internet was born, together with Hypertext Markup Language, used for describing text documents, we've got JavaScript - simple script language with C-like syntax for interacting and changing the structure of our documents. That was the first time we could move different elements across our browser's screens. This fact was noticed by Scott Porter, and back in 1998 he created the first JavaScript game library with a very original name: 'Game Lib'. He mostly focused on creating ports of old Nes or Atari games then, using animated gifs, but we can also find his Video Pool game in which he emulated rotation of a cue with a sprite of 150 different positions!

During the late 1990s and early 2000s, popularity of JavaScript increased and community created a first 'umbrella term' describing a collection of technologies used together to create interactive and animated web sites - DHTML (Dynamic HTML). Developers of the 'DHTML era' didn't forget about Porter's 'Game Lib', so in a couple of years Brent Silby presented 'Game Lib 2'. It is still possible to play lot of games created with that library on his Web site.

The DHTML era was a time when JavaScript games were as good as those made in Flash. Developers made a lot of DOM libraries useful for Game Development, like Beehive by Peter Nederlof with his outstanding Rotatrix (personally, I think that it is one of the best HTML game EVER), and developed the first really polished browser games - Jacob Sidelin, creator of 14KB Mario created the very first page dedicated to JavaScript games - http://www.javascriptgaming.com/.

And then, 2005 came. It was 'the year of AJAX'. Even if "AJAX" just stands for "Asynchronous JavaScript and XML", it was another 'umbrella term' describing methods, trends and technologies used to create new kind of web sites - WEB 2.0. Popularization of new JavaScript patterns introduced the ability to create multiplayer connections or even true emulators of old computers. Best examples of this time were "Freeciv" by Andreas Rosdal - port of *Sid Meier* Civilization, and Sarien.net by Martin Kool, emulator of old Sierra games.

And then came a new era in the history of Internet. It is called 'HTML5'!


Baku
G
G


HTML
5

2 JavaScript crash course

Introduction

This course is about HTML5 games, and it will rely on several JavaScript APIs: the canvas API for drawing, the requestAnimationFrame API for animating, the DOM API for dealing with inputs and user interaction, the Web Audio API for music and sound effects, and also on the WebSocket API for adding multi-participant features to the games. We might also use some APIs relevant to HTML5 persistence or Ajax if we need to load/save resources locally or remotely. Some games might also use the orientation/acceleration or geolocation APIs introduced by HTML5.

So yes, you will have to use JavaScript, . We indicated that "basic knowledge" in JavaScript is mandatory for this course, however in case we did not practice this langage for a long time, or in case you would like to enhance your knowledge right now, we wrote this document.

It is not about teaching JavaScript. There are lots of resources available on the Web, and even the W3C proposes a JavaScript training course that covers JavaScript and jQuery, the popular swiss knife library for JS developers.

This document will help you to start in good conditions. Remember that one great thing with the W3C courses is that everybody can help each other. Some of the classmates are really good in JavaScript and are usually very happy to help others when they encounter difficulties. Of course, I helped students to debug their JS code...

Michel, your trainer.

External resources

- http://www.codecademy.com/de/tracks/javascript-combined
- The book I used to learn JavaScript myself: http://www.wuala.com/lpuums/JavaScript/Object-Oriented%20JavaScript%20-%20Stoyan%20Stefanov.pdf?lang=fr
- Mozilla Developper Network has a JS guide too: https://developer.mozilla.org/en-US/docs/JavaScript/Guide
- You might have a look at w3schools.com but beware that this Web site is full of mistakes and untrue information, see http://w3fools.com/ if you don't believe me!

I also teach JS to my University students: see my JavaScript course slides (in French): http://mainline.essi.fr/JavaScriptSlides/index.html

Extracts from the forum posts (by students) during the previous version of the course:

Video tutorials at Treehouse which are very slick, this includes a basic Javascript course:

http://teamtreehouse.com/library/websites/javascript-foundations

Codecademy is also very good:

http://www.codecademy.com/tracks/javascript

I've also been through this Udemy video course which was useful for the basics too but maybe not quite so well structured:

https://secure.udemy.com/beginning-javascript/

And this Missing Manual book is good for jQuery, with a little bit of beginner JavaScript too:

http://shop.oreilly.com/product/0636920015048.do

I tend to steer clear of W3Schools but that's just a personal preference, haven't tried a full course from the start.

...

Might i add javascript 101 from jquery website :

http://learn.jquery.com/javascript-101/

Also i did followed the codeacademy JS course some months ago it's a very nice introduction for beginners. I do recommend for those with no JS knowledge as it starts from scratch.

What do you need? How to debug? How to catch errors?

We will not look at the JavaScript syntax here, but more at "JavaScript in the browser", how it works, how to start writing code, etc.

First of all, you need to find a way to debug your code and see errors.

For that you can use:

- Chrome dev tools (installed by default in Chrome, press F12 to display the dev tool console)
- The firebug extension for Firefox (go to the tools/additional modules and use the search dialog for "firebug", then install it). Once installed, press F12 to open the firebug console
- IE10 comes also with a pre-installed dev tool console (F12 again)
- For other browsers, please look for the related dev. tools

In these tools, you always have a "console tab" where errors will be displayed, or messages of your own (use the console.log(string) JavaScript function). In the console you can also type any JavaScript command.

Let's look at an example in the web based IDE jsbin.com : http://jsbin.com/ahahoc/2/edit

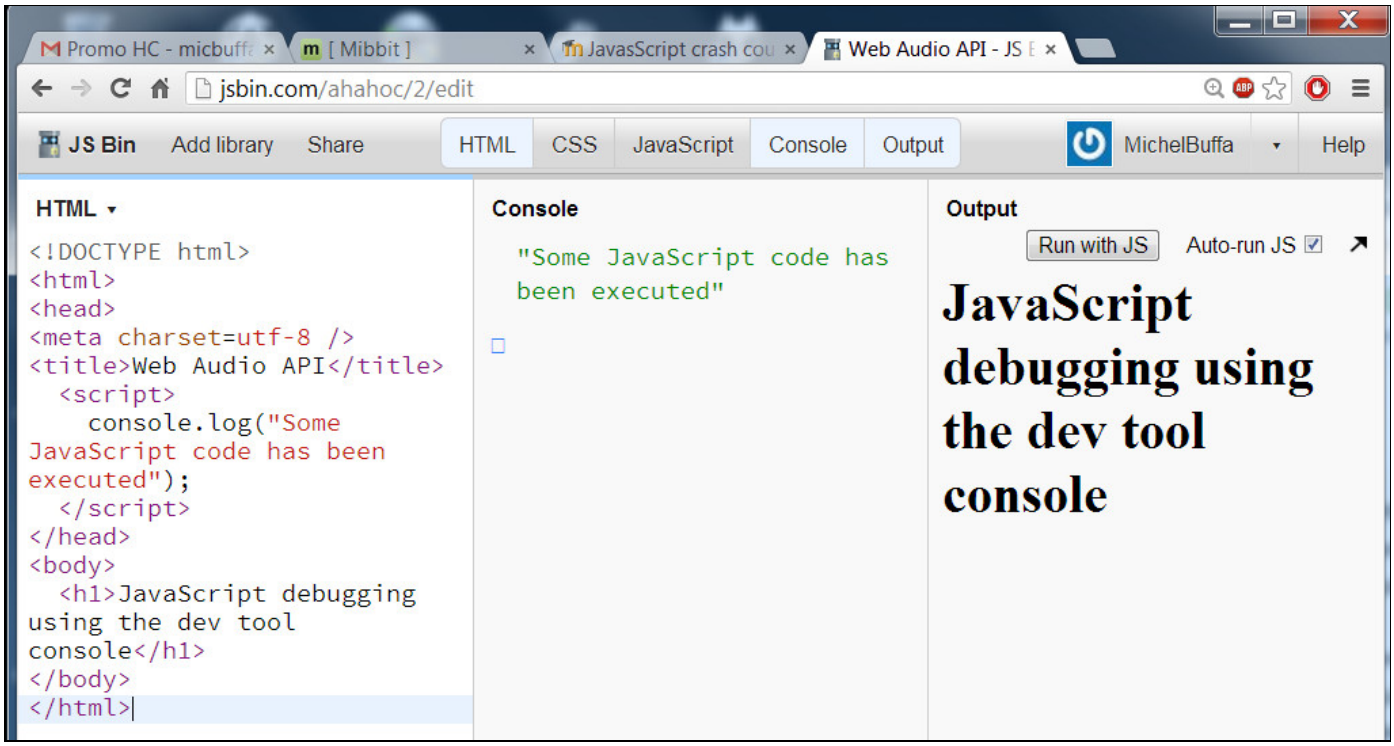
```
1  <!DOCTYPE html>
2  <html>
3    <head>
4    <meta charset=utf-8 />
5    <title>Web Audio API</title>
6    <script>
7     console.log("Some JavaScript code has been executed");
8    </script>
9    </head>
10   <body>
11     <h1>JavaScript debugging using the dev tool console</h1>
12   </body>
13 </html>
```

Well, this is the most simple way to add JavaScript code in an HTML page, using the <script>...</script> element. The code is executed sequentially when the page is loaded: the JavaScript code is executed before the browser could see the rest of the page, the H1 element, for example, does not exist in the Document

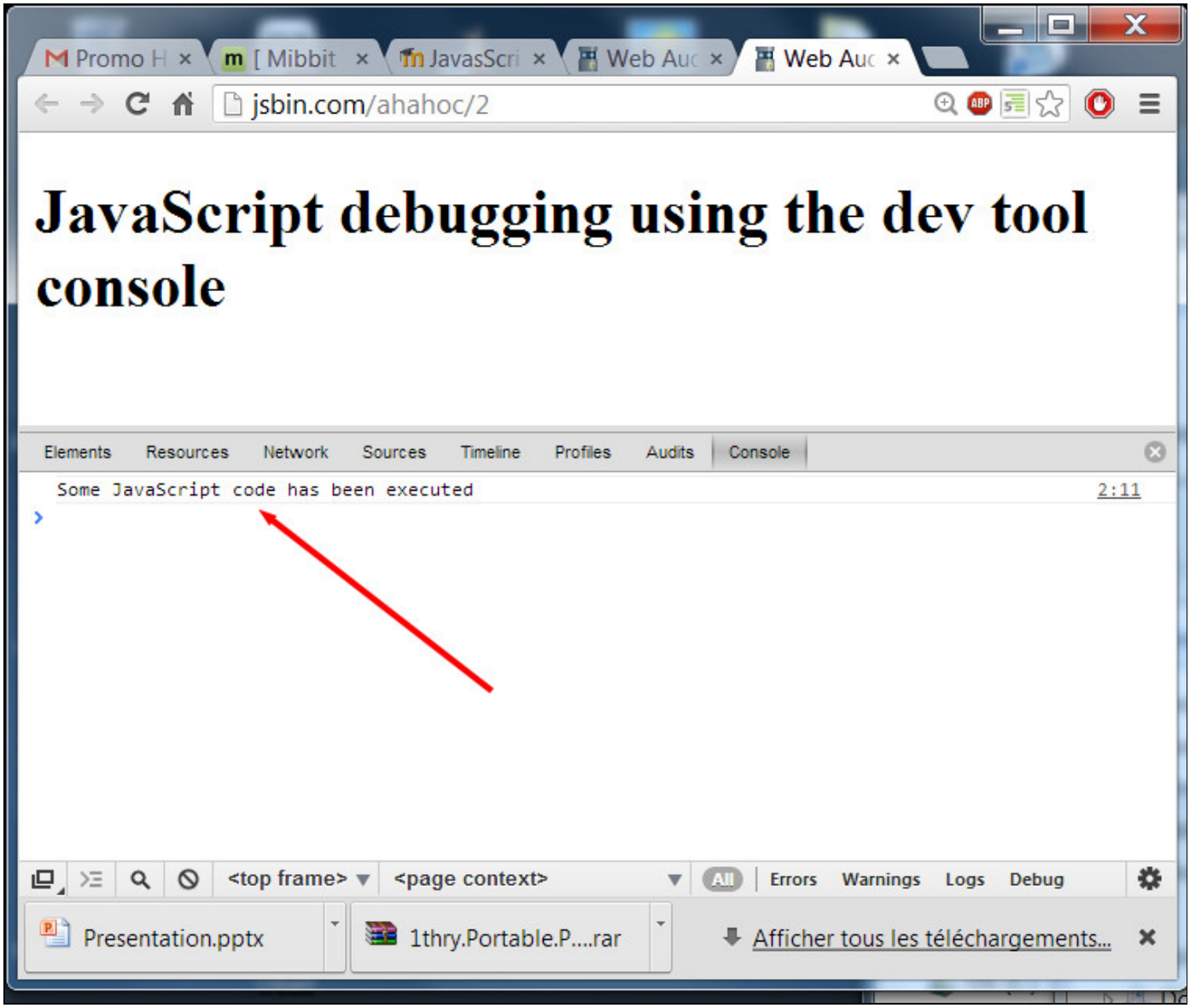
Object Model, and has not yet been displayed when the JavaScript code is executed.

The only line of code we have is console.log("Some JavaScript code has been executed");

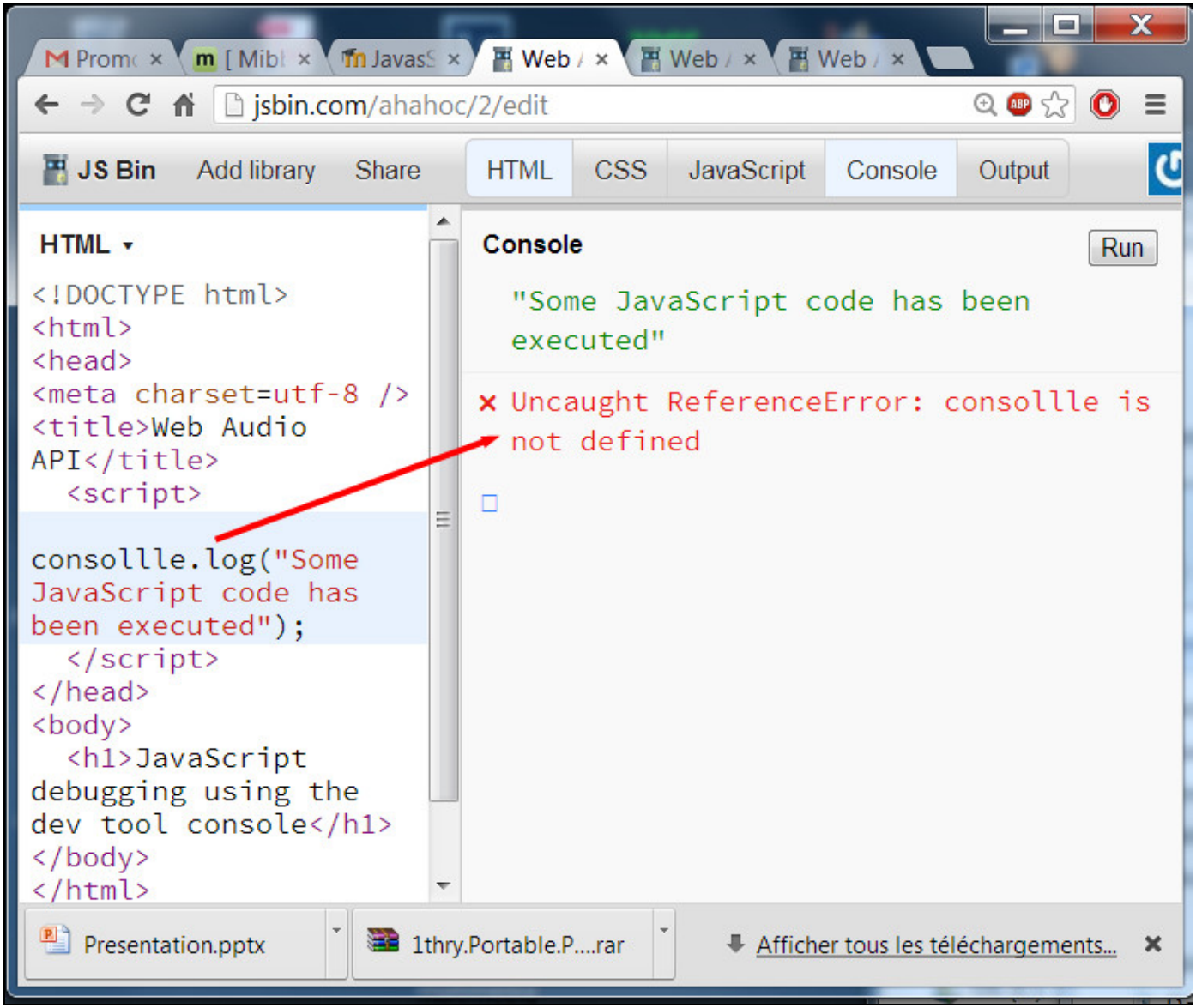
This means "display in the JavaScript console the message"... If we open the "console tab" in jsbin.com (that redirects all console.log() messages), and re-execute the page (just type a space at the end of a line, this will re-render the page and display the message in the console) we see in green the message in the console tab:



We can also use the "real dev tool console", and for this I recommend to run the application in a single window, not in the jsbin editor. Press the black arrow on the top right of the output window, this will render the page as a standalone Web page, then press F12. You should see:

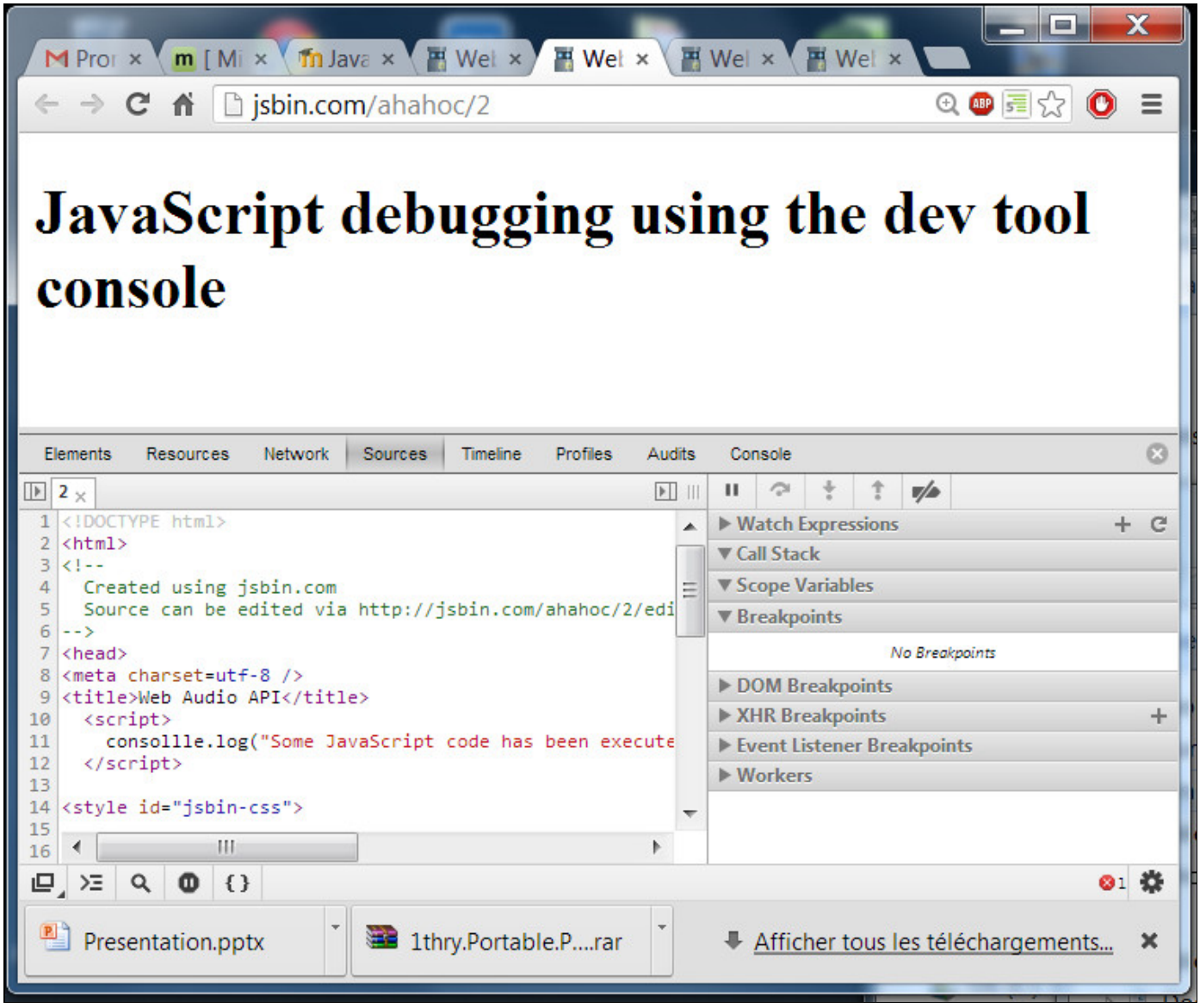


Ok, now, let's make an error, change console.log() into consollle.log(). Let's see what happens:



And if we run it standalone and use the dev tool console:

And if we click on the line number in the right, the dev tool shows the source code centered on the line that caused the error:



Without such tools, debugging JavaScript code is impossible. So you need to look at some basic tutorials on how to use Chrome Dev Tools, Firebug for Firefox or IE dev tools, etc... the way they work differ one from another but the principles remain the same.

Quick reminder of basic JavaScript

I assume that you already have an experience in JavaScript & HTML, so I will remind just couple of most important things you have to remember about during the course:

Methods are just the properties with functions assigned to them.

Here is a general example of a simple object:

```
var AwesomeObject = {
  property1: 'value',
  method: function(){
    alert('method!');
  }
}
```

Or another example:

```
var michel = {
   name: 'Michel Buffa',
   getName: function() {
      return name;
  }
}
```

Almost everything in JS is an object. So you can add properties to functions, etc.

Here are some examples:

```
var foo = function(){
   alert('I'm in the function');
}

foo.bar = function(){
   alert('all your object are belong to us');
}
```

There is no private scope in JavaScript, except the function scope

Functions can use global variables inside them:

```
var foo = 1; // global

var f = function(){
   alert('output: ' + foo); //output: 1
}

alert('output: ' + foo); //still 'output: 1'
```

But if you declare the variable inside the function, it will be visible only there.

```
var f2 = function(){
   var foo = 2;
   alert('output: ' + foo);
}

alert('output: ' + foo); //ReferenceError: foo is not defined
```

So to construct objects with private functions, we will use a pattern like this:

```
var NameOfThePseudoClass = function(){
   // private variables
   var privateVar1 = 0,
   privateVar2 = 1;

   // now private functions we need
   var privateMethod1 = function(args){
     //everything we need to be private in here
   }

      // actions that need to be executed on start are located here ('constructor like behavior'), for
example
      // call privateMethod1(...); or whatever...

      ...
     // public methods, visible outside of this function
     var publicMethod = function(){
       return privateVar1 + privatevar2;
     }

     // returning the object with some kind of 'public API', only with methods we want to use outside
     return {
        publicMethod : publicMethod //publicName : name of the function inside our scope
     }
}

//usage : create an instance
var instance = new NameOfThePseudoClass();

// Call a public method on the instance
instance.publicMethod();
```

We can manipulate the elements of the page structure using DOM techniques

In order to change or access some attributes on elements in the DOM of the HTML document, you first need to select the element(s), and then use JavaScript methods and the JavaScript DOM API to manipulate the element's attributes. You can also change the hierarchy by moving/adding/creating DOM elements. HTML5 added also new methods to the standard DOM level 2 JavaScript API, that will be studied in week 5 of the HTML5 course.

If this is our HTML document:

```
<head>
  <style>
    #span1 {
      width: 100px;
      height: 100px;
      background-color: #BADA55;
    }
  </style>
</head>
<body>
  <span id="span1"></span>
</body>
```

To change the background color of the span element, here is one way to do it:

var spanElement = document.getElementById('span1');
spanElement.style.backgroundColor = "#FF00FF';

REMEMBER! If you want to set or get number value, think about the units:

spanElement.style.width = 100; //WRONG!
spanElement.style.width = '100px'; //GOOD!

var width = spanElement.style.width; //WRONG! It will return '100px' string
var width = parseInt(spanElement.style.width); //WRONG!

In the above example, you have to provide the base of the numeric system you want to use - for example parseInt('071') returns 57, because of the leading '0' - it thinks that the number is in OCT system. Here is how to correctly do the parseInt(...)

var width = parseInt(spanElement.style.width, 10); //GOOD! It will return number 100

JavaScript code that selects DOM elements must be called after the page has been completely loaded

Indeed, when an HTML page is loaded, the JavaScript code is executed sequentially.

<canvas id='mycanvas'>.....</canvas>

```
<script>
  var c = document.getElementById("mycanvas"); var coontext = c.getContext('2d');
</script>
```

This will work, because the <canvas> is declared before the JavaScript code is executed.

```
<script>
  var c = document.getElementById("mycanvas"); var coontext = c.getContext('2d');
</script>

<canvas id='mycanvas'>.....</canvas>
```

This will not work, as c will be undefined.

This can lead to some inter-blocking techniques, and having to think at the location of the JavaScript code is really annoying. The solution is to run the code that access DOM elements in a function that is executed only once the page is completely loaded and the DOM ready.

There are multiple ways to do this, the most common one is to add a "onload" callback on the body element or to make the document listen to the "load" DOM event. Here is an example:

GOOD:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset=utf-8 />
    <title>This is a test</title>
    <script>
      function init() {
        var c = document.getElementById('mycanvas');
        var ctx = c.getContext('2d');
        ctx.fillRect(0, 0, 200, 200);
      }
  </script>
  </head>
  <body onload='init();'>
    <canvas id='mycanvas' width='200' height='200'></canvas>
  </body>
</html>
```

BAD (the next example is the bad version of the previous one):

```
<!DOCTYPE html>
<html>
  <head>
  <meta charset=utf-8 />
  <title>This is a test</title>
    <script>
            // mycanvas is defined AFTER this code is executed
      var c = document.getElementById('mycanvas');
      var ctx = c.getContext('2d');
      ctx.fillRect(0, 0, 200, 200);
    </script>
  </head>
  <body>
    <canvas id='mycanvas' width='200' height='200'></canvas>
  </body>
```

```
16 </html>
```

Other methods for selecting DOM elements

document.getElementsByTagName() - returns collection (indexed like an Array) of tags (like 'div' or 'span')
document.getElementsByClassName() - like before, but only with a given class.

Also, if an attribute is spelled with two words (like 'background-image'), you can access it from JavaScript using CamelCase notation ('backgroundImage'). Here is a full list of JavaScript references to CSS2 properties:

CSS Property	JavaScript Reference
background	background
background-attachment	backgroundAttachment
background-color	backgroundColor
background-image	backgroundImage
background-position	backgroundPosition
background-repeat	backgroundRepeat
border	border
border-bottom	borderBottom
border-bottom-color	borderBottomColor
border-bottom-style	borderBottomStyle
border-bottom-width	borderBottomWidth
border-color	borderColor
border-left	borderLeft
border-left-color	borderLeftColor
border-left-style	borderLeftStyle
border-left-width	borderLeftWidth
border-right	borderRight
border-right-color	borderRightColor
border-right-style	borderRightStyle
border-right-width	borderRightWidth
border-style	borderStyle
border-top	borderTop
border-top-color	borderTopColor

border-top-style	borderTopStyle
border-top-width	borderTopWidth
border-width	borderWidth
clear	clear
clip	clip
color	color
cursor	cursor
display	display
filter	filter
font	font
font-family	fontFamily
font-size	fontSize
font-variant	fontVariant
font-weight	fontWeight
height	height
left	left
letter-spacing	letterSpacing
line-height	lineHeight
list-style	listStyle
list-style-image	listStyleImage
list-style-position	listStylePosition
list-style-type	listStyleType
margin	margin
margin-bottom	marginBottom
margin-left	marginLeft
margin-right	marginRight
margin-top	marginTop
overflow	overflow
padding	padding
padding-bottom	paddingBottom
padding-left	paddingLeft
padding-right	paddingRight

padding-top	paddingTop
page-break-after	pageBreakAfter
page-break-before	pageBreakBefore
position	position
float	styleFloat
text-align	textAlign
text-decoration	textDecoration
text-decoration: blink	textDecorationBlink
text-decoration: line-through	textDecorationLineThrough
text-decoration: none	textDecorationNone
text-decoration: overline	textDecorationOverline
text-decoration: underline	textDecorationUnderline
text-indent	textIndent
text-transform	textTransform
top	top
vertical-align	verticalAlign
visibility	visibility
width	width
z-index	zIndex

If you want to create a new element, and append it to existing one, use this similar method:

var newElement = document.createElement('div');
spanElement.appendChild(newElement);

Other helpful tools:

1. JsLint - code quality tool. Paste your code inside, and it will show you where you made errors.
2. JsBeautifier - paste non formated JavaScript code in there to make it more readable. It will help you during this course, because sometimes the snippets I create display without indents.
3. JsFiddle - sandbox for testing fragments of code, like jsbin.com
4. JsPerf - performance playground. If you don't know it your solution is fast enough, you can test it there. It has also lot of examples made by other developers: JsPerf Test cases.
5.

3 Elements and APIs that will be useful for writing games

New HTML5 elements useful in game development

Drawing

<canvas>

The <canvas> is a new HTML element described as "a resolution-dependent bitmap canvas which can be used for rendering graphs, game graphics, or other visual images on the fly." It's a rectangle included in your page where you can draw using scripting with JavaScript. It can for instance be used to draw graphs, make photo compositions or do animations. This element consists of a drawable region defined in HTML code with height and width attributes.
You can have multiple canvas elements on one page, even stacked one onto another, like transparent layers. Each will be visible in the DOM tree and has it's own state independent of the others. It behaves like regular DOM element.

The canvas has a rich JavaScript API for drawing all kinds of shapes, we can draw wireframe of filled shapes and set several properties such as color, line width, patterns, gradients, etc. It also supports transparency and pixel level manipulations. Today it is supported by all browsers, on desktop or mobile phones, and on most devices it will take advantage of hardware acceleration.

It's for sure most important new element in the HTML5 spec from game developer's point of view, so we will discuss it with more details later, in next lessons.

Animating

requestAnimationFrame API

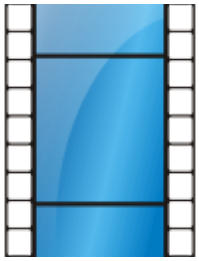
The requestAnimationFrame API targets 60 frames/s animation in canvases. This API is quite simple and comes also with a high resolution timer. Animation at 60 frames/s is often easy to obtain with simple 2D games, on major desktop computers. This is the preferred way for performing animation as the browser will take care of not performing the animation when the canvas is not visible, thus saving up cpu.

Videos and animated textures

The <video> element

HTML5 video is an element introduced in the HTML5 specification for the purpose of playing videos or movies, partially replacing the object element. Usability of the element is lowered because of lack of agreement between browser vendors as of which video formats should be supported. The API is close to the one of the <audio> element.

By combining the capabilities of the <video> element with a canvas, it's possible to manipulate video data



in real time to incorporate a variety of visual effects to the video being displayed, on on the contrary, to use images from videos as "animated textures" on graphic objects.

Audio (streamed audio and real time sound effects)

The <audio> element

<audio> is a HTML element which was introduced to give a consistent API for playing *streamed* sounds in browsers. File format support differs from one browser to another, but MP3 works on nearly all browsers today. Unfortunately the <audio> tag is only for streaming compressed audio, so it consumes cpu resources, and is not adapted for sound effects where you would like to change the playing speed or add real time effects like reverb or doppler. For this we will prefer the new Web Audio API.

The Web Audio API

This is a 100% javascript API designed for working in real time with uncompressed sound samples or for generating procedural music. Sound samples will need to be loaded in memory and decompressed prior to being used. Up to 12 sound effects are provided natively by browsers that supports the API (all major ones except IE, but support has been announced by Microsoft for IE12).

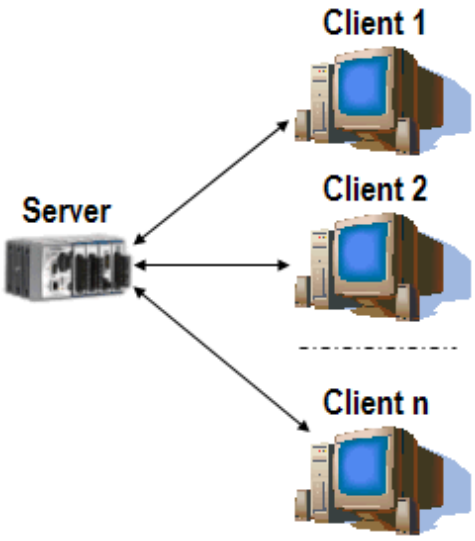
Interacting

User inputs will rely on several APIs, some are well established like the DOM API that we will use for keyboard, touch or mouse inputs. There is also a working draft GamePad API that is already implemented by some browsers. We will look into it also in that course.

Multi Participant features

WebSockets

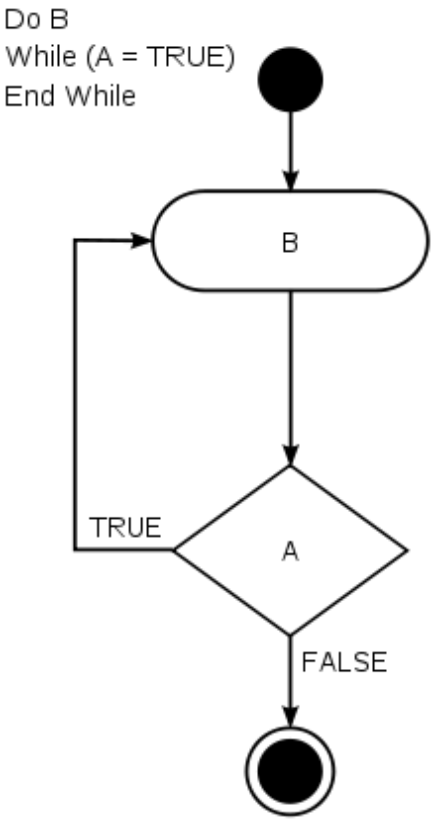
Using the WebSockets technology (that is not in the HTML5 specification but comes from WebRTC) you can create two-way communication sessions between several browsers and a server. The WebSocket API provides means for sending messages to a server and receive event-driven responses without having to poll the server for a reply.



4 The "Game loop"

Different implementations of the 'Main Game Loop'

Introduction



The "game loop" is the main component of any game. It separates the game logic and the visual layer from a user's input and actions.

Traditional applications respond to user input and do nothing without it - word processor formats text as a user types. If the user doesn't type anything, the word processor is waiting for an actions.

It looks a way different in games: a game must continue to operate regardless of a user's input.

The game loop allows this - it is computing events in our game all the time. Even if the user doesn't make any actions, the game will move the enemies, resolve collisions, play sounds and draw graphics as fast as possible.

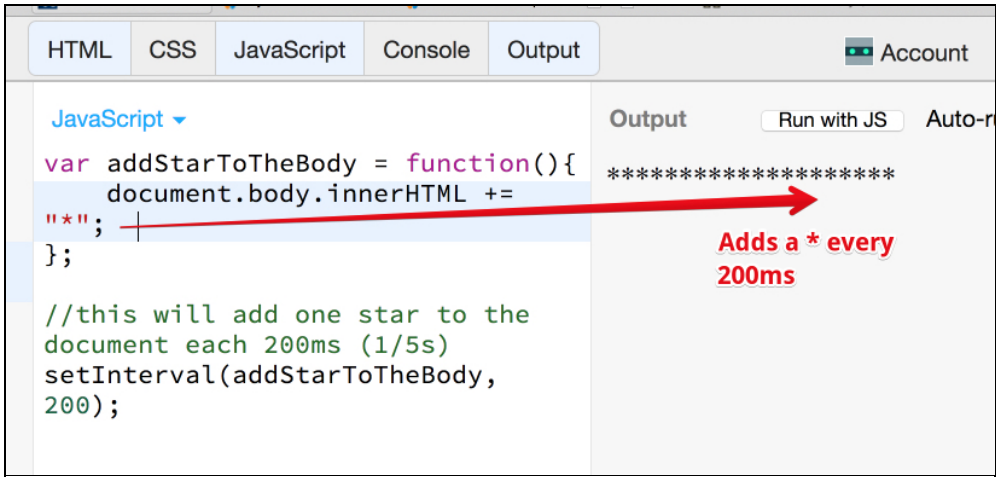
There exists different ways to perform animation with JavaScript. Let's have a quick look at them.

Performing animation using the JavaScript setInterval(...) function

- Syntax: **setInterval(function, ms);**

The setInterval function calls a function or evaluates an expression at specified intervals of time (in milliseconds), and returns a unique id of the action. You can always stop that by calling the **clearInterval(id)** function with the interval identifier as an argument.

Example you can try it online at: http://jsbin.com/yuhule/3/edit (open the html, javascript and output tabs):



Here is the source code:

```
var addStarToTheBody = function(){
    document.body.innerHTML += "*";
};

//this will add one star to the document each 200ms (1/5s)
setInterval(addStarToTheBody, 200);

```

NOTE: Even if it can evaluate the expressions, you should never do this. Use anonymous functions instead.

WRONG:

```
setInterval('addStarToTheBody()', 200);
setInterval('document.body.innerHTML += "*";', 200);
```

GOOD:

```
setInterval(function(){
    document.body.innerHTML += "*";
}, 200);
```

REMEMBER: You should avoid evaluations (providing strings to eval(), setInterval() & setTimeout() functions). You can achieve the same effect in better ways. ALWAYS!

Using setTimeout() instead of setInterval()

One thing you should always remember about using setInterval - if we set number of milliseconds at let's say 200, it will call our game loop function EACH 200ms, *even if the previous one is not yet finished*. Thats why we can use another function:

- Syntax: setTimeout(function, ms);

This function works like setInterval with one little difference - it calls your function **AFTER** given amount of time.

Example of use: http://jsbin.com/zudebaxoze/3/edit (open javascript, console and output tabs).

```
var addStarToTheBody = function(){
    document.body.innerHTML += "*";
    // calls again itself AFTER 200ms
    setTimeout(addStarToTheBody, 200);
};

// calls the function AFTER 200ms
setTimeout(addStarToTheBody, 200);

```

This example will work like that one from the previous example. *It is a way better*, because the timer waits for the function to finish everything inside before calling it back again.

For several years, setTimeout used to be the best and most popular JavaScript implementation of game loops. Till Mozilla presents the requestAnimationFrame API and it became the reference W3C standard API for game animation.

Using the requestAnimationFrame API

When you use timeouts or intervals in your animation, the browser doesn't have any information about your intentions - do you want to repaint the DOM structure ora canvas during every loop? Or maybe you just want to make some calculations or requests a couple of times a second? Because of that, it is really hard for the browser's engine to optimize your loop.

And since you want to repaint your game (move the characters, animate sprites, etc) on each frame, Mozilla and other contributors/developers proposed a new approach they called **requestAnimationFrame**.

It will help your browser to optimize all the animations on the screen, no matter if you use Canvas, Dom or WebGL. Also, if you're running the animation loop in a browser tab that is not visible, the browser won't keep it running.

Basic usage, online example at: http://jsbin.com/gixepe/2/edit

```
1  window.onload = function init() {
2     // called after the page is entirely loaded
3     requestAnimationFrame(mainloop);
4  };
5
6  function mainloop(time) {
7     document.body.innerHTML += "*";
8
9     // call back itself every 60th of second
10    requestAnimationFrame(mainloop);
11 }
```

Notice that calling requestAnimationFrame(mainloop) at line 10, "asks the browser to call the mainloop function every 16,6 ms", this corresponds to 1/60th if a second. This target may be hard to reach, the animation loop content may take more than that, or the scheduler may be a bit late or in advance. Many"real action games" perform what we call "time based animation", we will study this later in the course... but for this, we need an accurate timer that will tell us the elapsed time tween each animation frames. Depending on this time we can compute the distances each object on the screen must achieve in order to move at a given speed, independently of the cpu or gpu of the computer or mobile device that is running the game.

The timestamp parameter of the mainloop function is exactly useful for that: it gives a high resolution time.

Notice that "old browsers" implemented some prefixed, experimental versions of the API and you might encounter in tutorials on the web some piece of code that use requestAnimationFrame with a polyfill that will ensure the examples work on any browser, including the ones that do not support this API at all (falling back to setTimeout)

The most famous polyfill has been written by Paul Irish from the jQuery team. He wrote this polyfill to simplify the usage of requestAnimationframe in different browsers:

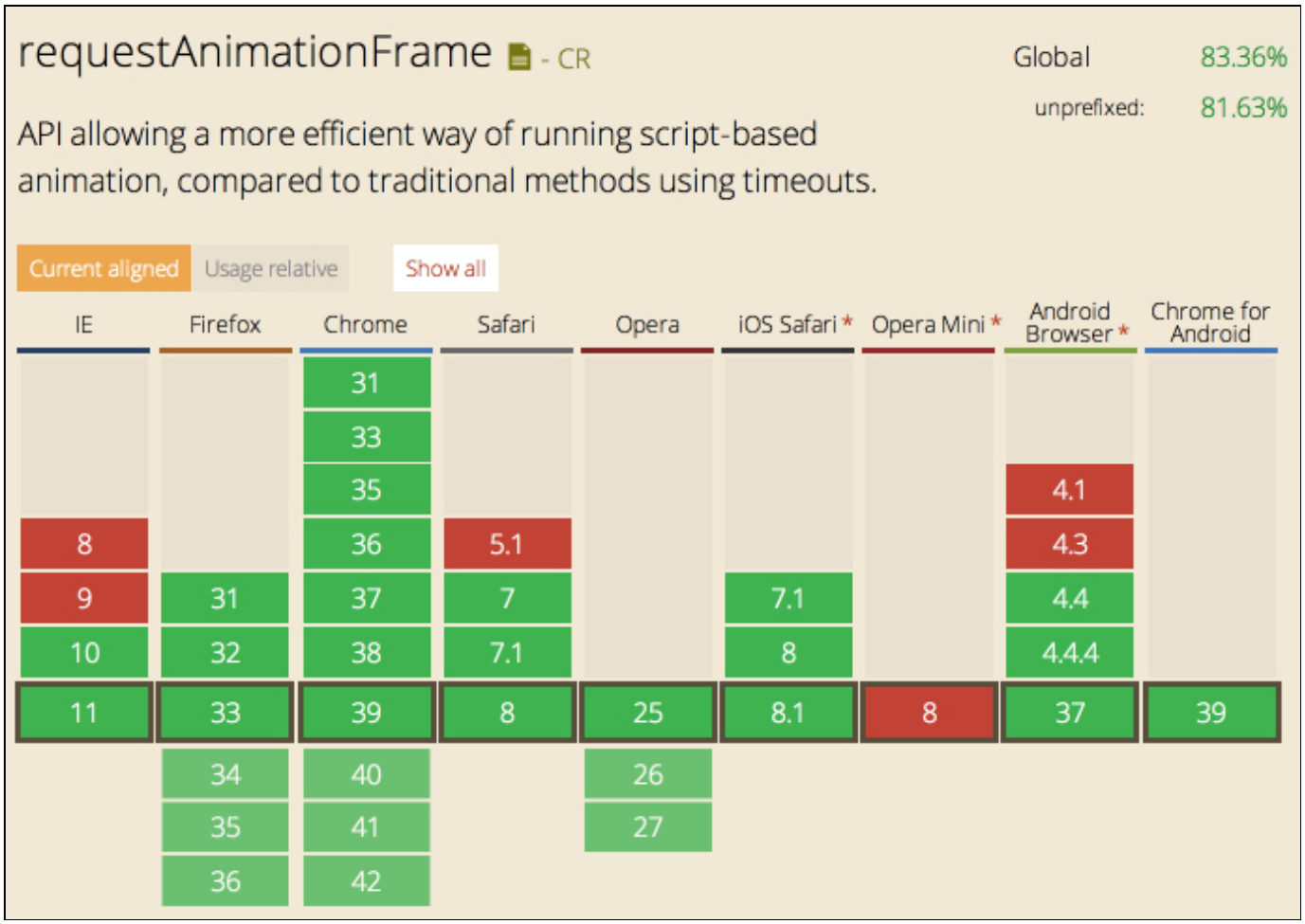
```
1  // shim layer with setTimeout fallback
2  window.requestAnimFrame = (function(){
3  return window.requestAnimationFrame ||
4  window.webkitRequestAnimationFrame ||
5  window.mozRequestAnimationFrame ||
6  window.oRequestAnimationFrame ||
7  window.msRequestAnimationFrame ||
8  function(/* function */ callback, /* DOMElement */ element){
9      window.setTimeout(callback, 1000 / 60);
10   };
11 })();
```

So according to our last example, it will look like this using requestAnimationFrame:

```
// shim layer with setTimeout fallback
window.requestAnimFrame = (function(){
return window.requestAnimationFrame ||
window.webkitRequestAnimationFrame ||
window.mozRequestAnimationFrame ||
window.oRequestAnimationFrame ||
window.msRequestAnimationFrame ||
function(/* function */ callback, /* DOMElement */ element){
window.setTimeout(callback, 1000 / 60);
};
})();

window.onload = function init() {
   requestAnimFrame(mainloop);
};

function mainloop(time) {
   document.body.innerHTML += "*";

   // call back itself every 60th of second
   requestAnimFrame(mainloop);
}

```

Notice that this polyfill defines a function named requestAnimFrame (instead of the standard requestAnimationFrame).

As today, the support for the standard API is very good with modern browsers, we will not use this polyfill in our examples. If you would like to target also "old browsers", just adapt your code to this polyfill, it's just a matter of changing two lines of code and inserting the JS polyfill.



RequestAnimationFrame support (November 2014)

A Game Framework squeletton that uses requestAnimationFrame

A game framework squeletton that uses requestAnimationFrame

Here is a (very) small squeletton:

```
var GF = function(){

    var mainLoop = function(time){
        //main function, called each frame
        requestAnimationFrame(mainLoop);
    };

    var start = function(){
        requestAnimationFrame(mainLoop);
    };

    //our GameFramework returns a public API visible from outside its scope
    return {
        start: start
    };
};
```

With this squeletton it's very easy to create a new game instance:

```
var game = new GF();

// Launch the game, start the animation loop etc.
game.start();
```

Let's put something into the mainLoop function, and check if it works.

Ty this online example with a new mainloop: http://jsbin.com/kafehi/3/edit (open javascript and output tabs). It should display at 60 frames/s a random number right in the body of the document. We're far from a real game yet, but we're improving our game engine :-)

```
var mainLoop = function(time){
   //main function, called each frame
   document.body.innerHTML = Math.random();

   // call the animation loop every 1/60th of second
   requestAnimationFrame(mainLoop);
};

```

Now we need to count frames per seconds.

That's a classic: every game need to have a FPS measuring function. The principle is simple: count the time elapsed by adding deltas in the mainloop. If the sum of the deltasis greater or equal to 1000, then 1s elapsed. If at the same time we count the number of frames that have been drawn, then we have the number of frames per second. Remember it should be around 60 frames/second.

Here is the code we added to our game engine, for measuring FPS (try it online: http://jsbin.com/kafehi/5/edit)

```
    // vars for counting frames/s, used by the measureFPS function
    var frameCount = 0;
    var lastTime;
    var fpsContainer;
```

```
    var fps;

    var measureFPS = function(newTime){

         // test for the very first invocation
         if(lastTime === undefined) {
           lastTime = newTime;
           return;
         }

        //calculate the difference between last & current frame
        var diffTime = newTime - lastTime;

        if (diffTime >= 1000) {
            fps = frameCount;
            frameCount = 0;
            lastTime = newTime;
        }

        //and display it in an element we appended to the
        // document in the start() function
       fpsContainer.innerHTML = 'FPS: ' + fps;
       frameCount++;
    };

```

And we will call the function from inside the animation loop, passing it the current time, given by the high resolution timer that comes with the requestAnimationFrame API:

```
    var mainLoop = function(time){
        //main function, called each frame
        measureFPS(time);

        // call the animation loop every 1/60th of second
        requestAnimationFrame(mainLoop);
    };

```

And the DIV element used to display FPS on the screen is created in this example by the start() function:

```
    var start = function(){
        // adds a div for displaying the fps value
        fpsContainer = document.createElement('div');
        document.body.appendChild(fpsContainer);

        requestAnimationFrame(mainLoop);
    };

```

Hack : achieving more than 60 frames/s ? It's possible but avoid except in private hackers' circles !

Hack: achieving more than 60 frames/s ? It's possible but avoid except in private hackers' circles !

We know also methods of implementing loops in JavaScript and achieving even more than 60fps (this is the limit using requestAnimationFrame).

My favorite hack uses onerror callback on <img> element like this:

```
function loop(callback){
```

```
    var img = new Image;
    img.onerror = callback;
    img.src = 'data:image/png,' + Math.random();
}
```

What we are doing in here, is creating new Image on each frame, and providing invalid data as a source of the image. The Image then cannot be displayed properly, so the browser calls the onerror event handler.

Funny hey, you can try this and check the number of FPS displayed: http://jsbin.com/temohe/2/edit

We just changed the mainLoop to this code:

```
    var mainLoop = function(){
        //main function, called each frame

        measureFPS(+(new Date()));

        // call the animation loop every LOTS of second
        var img = new Image();
        img.onerror = mainLoop;
        img.src = 'data:image/png,' + Math.random();
    };

```

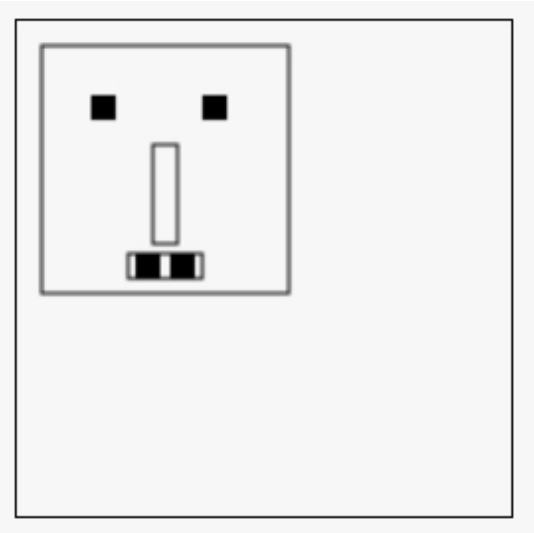
5 Is this really a course about games ? Where are the graphics ???

Calm down, we will study in details how to draw in the HTML5 canvas. So far we just played with "basic concepts", but we can see you can't wait to draw something, and move shapes on the screen :-)

Let's see rapidly the basic concepts with the canvas, we will look at them deepier the next week. Today, we will just have "a taste of the HTML5 canvas".

HTML5 canvas basic usage

Online example: draw a monster in a canvas. You can try it online at: http://jsbin.com/pagipi/2/edit



HTML code (declaration of the canvas):

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <meta charset="utf-8">
5    <title>JS Bin</title>
6  </head>
7  <body>
8
9    <canvas id="myCanvas" width="200" height="200"></canvas>
10
11 </body>
12 </html>
```

The canvas declaration it as line 9. Give it a width and a height, but unless you add some CSS properties, you will not see it on the screen: it's transparent!

CSS to reveal the canvas:

```
1 canvas {
2   border: 1px solid black;
3 }
```

And here is a good practice on how to use the canvas:

1. In a function called AFTER the page is fully loaded (and the DOM ready), get a pointer to the canvas

node in the DOM,
2. Then get a 2D graphic context for this canvas (the context is an abject we will use to draw on the canvas, to set global properties like color, gradients, patterns, line width etc...
3. Then only draw something. And do not forget to use global variables for the canvas and context objects. I also recommend to keep somewhere the width and height of the canvas. This might be useful later.
4. Draw something.

Here is the JavaScript code that corresponds to this good practice:

```
1  // useful to have them as global variables
2  var canvas, ctx, w, h;
3
4
5  window.onload = function init() {
6      // called AFTER the page has been loaded
7      canvas = document.querySelector("#myCanvas");
8
9      // often useful
10     w = canvas.width;
11     h = canvas.height;
12
13     // important, we will draw with this object
14     ctx = canvas.getContext('2d');
15
16     // ready to go !
17     drawMyMonster();
18 };
19
20 function drawMyMonster() {
21     // draw a big monster !
22     // head
23     ctx.strokeRect(10, 10, 100, 100);
24
25     // eyes
26     ctx.fillRect(30, 30, 10, 10);
27     ctx.fillRect(75, 30, 10, 10);
28
29     // nose
30     ctx.strokeRect(55, 50, 10, 40);
31
32     // mouth
33    ctx.strokeRect(45, 94, 30, 10);
34
35    // teeth
36    ctx.fillRect(48, 94, 10, 10);
37    ctx.fillRect(62, 94, 10, 10);
38 }
```

In this small example we used the context object to draw a monster using the default color (black) and wireframe and filled modes:

- ctx.fillRect(x, y, width, height): draw a rectangle whose top left corner is at (x, y) and whose size is specified by the width and height parameters.
- ctx.strokeRect(x, y, width, height): same but il wireframe mode.

Moving more easily the monster from the previous example

In the drawMonster() function, everything is hardcoded, but how could we draw easily this monster somewhere else? The answer is: use 2D geometric transformation like translate, rotate, scale, etc... the context object provides such methods. We will look at them in details the next week. So far, we will just use the translate(x y) method. And if we change the coordinate system (this is what a call to translate does), it is always a good practice to save the previous context when entering a function that may change it, and

restore it at the end of the function.

So here is our new version of the drawMonster function. We added two parameters for specifying the (x, y) position of the monster:

```
function drawMyMonster(x, y) {
    // draw a big monster !
    // head

    // save the context
    ctx.save();

    // translate the coordinate system, draw relative to it
    ctx.translate(x, y);

    // (0, 0) is the top left corner of the monster.
    ctx.strokeRect(0, 0, 100, 100);

    // eyes
    ctx.fillRect(20, 20, 10, 10);
    ctx.fillRect(65, 20, 10, 10);

    // nose
    ctx.strokeRect(45, 40, 10, 40);

    // mouth
   ctx.strokeRect(35, 84, 30, 10);

   // teeth
   ctx.fillRect(38, 84, 10, 10);
   ctx.fillRect(52, 84, 10, 10);

   // restore the context
   ctx.restore();
}
```

And the call to the drawMonster is also a little bit different as we now pass the (x, y) position of the top left corner of the monster as parameters:

```
    // Try to change the parameter values to move
    // the monster
    drawMyMonster(10, 10);
```

You can try this new version (and change the parameters to see the monster move): http://jsbin.com/pagipi/3/edit

Animate the monster, include it in our game engine...

Ok, now that we know how to move this monster, let's integrate it in our game engine:

1. Add the canvas to the HTML page,
2. Add the content of the init function to the start() function of the engine, add a few global variables,
3. call the drawMonster function from the mainLoop,
4. Add a random displacement to the x, y position of the monster to see it move...
5. Do not forget to clear the canvas before drawing again, this is done by the ctx.clearRect(x, y, width, height) function.

You can try this version online here: http://jsbin.com/kafehi/7/edit

HTML code:

```
<!DOCTYPE html>
```

```
<html>
<head>
  <meta charset="utf-8">
  <title>JS Bin</title>
</head>
<body>
  <canvas id="myCanvas"  width="200" height="200"></canvas>
</body>
</html>
```

JavaScript complete code:

```
// Inits
window.onload = function init() {
  var game = new GF();
  game.start();
};


// GAME FRAMEWORK STARTS HERE
var GF = function(){
    // Vars relative to the canvas
    var canvas, ctx, w, h;

    // vars for counting frames/s, used by the measureFPS function
    var frameCount = 0;
    var lastTime;
    var fpsContainer;
    var fps;

    var measureFPS = function(newTime){

         // test for the very first invocation
         if(lastTime === undefined) {
           lastTime = newTime;
           return;
         }

        //calculate the difference between last & current frame
        var diffTime = newTime - lastTime;

        if (diffTime >= 1000) {
            fps = frameCount;
            frameCount = 0;
            lastTime = newTime;
        }

        //and display it in an element we appended to the
        // document in the start() function
       fpsContainer.innerHTML = 'FPS: ' + fps;
       frameCount++;
    };

     // clears the canvas content
     function clearCanvas() {
       ctx.clearRect(0, 0, w, h);
     }

     // Functions for drawing the monster and maybe other objects
     function drawMyMonster(x, y) {
       // draw a big monster !
       // head

       // save the context
       ctx.save();

       // translate the coordinate system, draw relative to it
       ctx.translate(x, y);

       // (0, 0) is the top left corner of the monster.
```

```
59          ctx.strokeRect(0, 0, 100, 100);
60
61          // eyes
62          ctx.fillRect(20, 20, 10, 10);
63          ctx.fillRect(65, 20, 10, 10);
64
65          // nose
66          ctx.strokeRect(45, 40, 10, 40);
67
68          // mouth
69          ctx.strokeRect(35, 84, 30, 10);
70
71          // teeth
72          ctx.fillRect(38, 84, 10, 10);
73          ctx.fillRect(52, 84, 10, 10);
74
75        // restore the context
76        ctx.restore();
77      }
78
79      var mainLoop = function(time){
80          //main function, called each frame
81          measureFPS(time);
82
83          // Clear the canvas
84          clearCanvas();
85
86          // draw the monster
87          drawMyMonster(10+Math.random()*10, 10+Math.random()*10);
88
89          // call the animation loop every 1/60th of second
90          requestAnimationFrame(mainLoop);
91      };
92
93      var start = function(){
94          // adds a div for displaying the fps value
95          fpsContainer = document.createElement('div');
96          document.body.appendChild(fpsContainer);
97
98          // Canvas, context etc.
99          canvas = document.querySelector("#myCanvas");
100
101         // often useful
102         w = canvas.width;
103         h = canvas.height;
104
105         // important, we will draw with this object
106         ctx = canvas.getContext('2d');
107
108         // start the animation
109         requestAnimationFrame(mainLoop);
110     };
111
112     //our GameFramework returns a public API visible from outside its scope
113     return {
114         start: start
115     };
116 };
117
```

Notice that we now start the game engine in a window.onload function, only when the page has been loaded. We also moved 99% of the init() method we wrote in the previous example into the start() method of the game engine, and added the canvas, ctx, w, h variables as variables global to the game framework object.

Finally we added a call to the drawMonster function in the mainloop, with some randomness in the parameters, like that the monster is drawn with an offset between [0, 10] at each frame of animation.

And we clear the canvas content before drawing the current frame content.

If you try the example, you will see a trembling monster. The canvas is cleared + monster drawn at random positions 60 times per second!

In the next part of this week's course we'll see how to interact with it using the mouse or the keyboard.

6 User interaction and events handling

Input & output: how events work in web apps & games?

Introduction / event management in JavaScript

There is no input or output in JavaScript. We treat events made by user as an input, and we manipulate DOM structure as output. Most of the times in games, we will change state variables of moving objects like position or speed of an alien ship, and the animation loop will take care of these variables to move the objects.

In any cases the events are called DOM events, and we use the DOM APIs to create event handlers.

There are three ways to manage events in the DOM structure. You can attach event inline in you HTML code like this:

Declare event handlers in the HTML code

```
1 <div id="someDiv" onclick="alert('clicked!')"> content of the div </div>
```

This is not the recommended way to handle events, even if it's very easy to use. Indeed, It works now, but it's deprecated and will probably be abandoned in a future. Mixing 'visual layer' (HTML) and 'logic layer' (JavaScript) in one place is really bad and causes lot of problems during development.

Add an event handler to an HTML element in JavaScript

Here is an example:

```
1 document.getElementById('someDiv').onclick = function() {
2     alert('clicked!');
3 }
```

This method is ok, but you will not be able to attach several listener functions. If you need to do this, the preferred version is the next one.

Register a callback to the event listener with the addEventListener method

This is how we can do that

```
1 document.getElementById('someDiv').addEventListener('click', function() {
2     alert('clicked!');
3 }, false);
```

Note that the third parameter describes if the callback has to be called during the captured phase. Tt is not important for now, just set it to false...

The DOM event that is passed to the event listener function

When you create an EventListener and attach it to an element, it will provide an event object as a parameter to your callback, just like this:

```
1 element.addEventListener('click', function(event) {
2    // now you can use event object inside the callback
3 }, false);
```

Depending on the type of event you are listening to, we will use different properties from the event object in order to get useful information like: "what keys have been pressed down?", "what is the position of the mouse cursor?", "which mouse button is down?", etc... We will cover now how to deal with the keyboard and with the mouse. Some experimental APIs like the gamePad API are on their way and supported by some browsers.

Dealing with key events

This has been sort of nightmare for years, as different browsers had different of handling key events and key codes (read this if you are found of JavaScript archeology: http://unixpapa.com/js/key.html)... fortunately it's much better today and we can rely on methods that should work on any browser less than four years old.

When you listen to keyboard related events (keydown or keyup), the event function will contain the code of the key that fired the event. Then it is possible to test what key has been pressed or released, like this:

```
1 window.addEventListener('keydown', function(event) {
2     if (event.keyCode === 37) {
3         //left arrow was pressed
4     }
5 }, false);
```

At line 2, 37 is the key code that corresponds to the left arrow. It might be difficult to know how the codes, so here is a quick reminder...

You can try key codes with this interactive example: http://www.asquare.net/javascript/tests/KeyCode.html

Here is a list of keyCodes (taken from: http://css-tricks.com/snippets/javascript/javascript-keycodes/)

Key	Code
backspace	8
tab	9
enter	13
shift	16
ctrl	17
alt	18
pause/break	19
caps lock	20
escape	27
(space)	32
page up	33
page down	34
end	35
home	36
left arrow	37
up arrow	38
right arrow	39
down arrow	40
insert	45
delete	46
0	48
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57
a	65
b	66
c	67
d	68

Key	Code
e	69
f	70
g	71
h	72
i	73
j	74
k	75
l	76
m	77
n	78
o	79
p	80
q	81
r	82
s	83
t	84
u	85
v	86
w	87
x	88
y	89
z	90
left window key	91
right window key	92
select key	93
numpad 0	96
numpad 1	97
numpad 2	98
numpad 3	99
numpad 4	100
numpad 5	101
numpad 6	102
numpad 7	103

Key	Code
numpad 8	104
numpad 9	105
multiply	106
add	107
subtract	109
decimal point	110
divide	111
f1	112
f2	113
f3	114
f4	115
f5	116
f6	117
f7	118
f8	119
f9	120
f10	121
f11	122
f12	123
num lock	144
scroll lock	145
semi-colon	186
equal sign	187
comma	188
dash	189
period	190
forward slash	191
grave accent	192
open bracket	219
back slash	220
close braket	221
single quote	222

Keep in a JavaScript object the list of key that are pressed at any moment

In a game, often we need to check what keys are down at a very high frequency, typically from inside the game loop, that is running up to 60 times per second. If a spaceship is moving left, there are chances you

are keeping the left arrow down, and if it's firing missiles at the same time you must be also press the spacebar like a maniac and also press the shift key to release smart bombs. Sometimes these three keys might be down at the same time, and the game loop will have to take these three keys into account: move the ship left, release a new missile if the previous one is out of the screen or if it reached a target, launch a smart bomb if conditions are ok, etc...

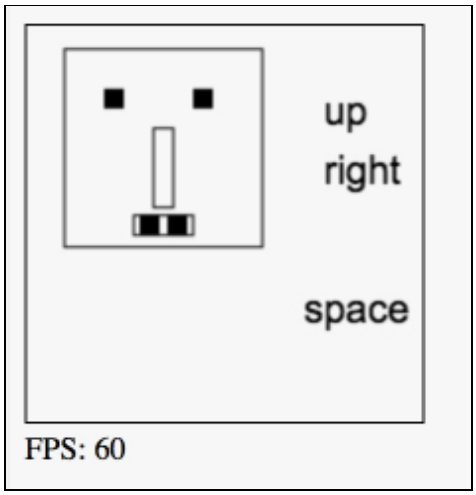
The typical method is: store in an object the list of the keys (or mouse button or whatever game pad button...) that are up or down at a given time. For our small game engine we will call this object "inputStates".

We will update its content inside the different input event listeners, and we will check its value from inside the game loop, 60 times/s.

So here are the things we have changed in our small game engine prototype (yet, far from finished):

1. We added an empty inputStates object as a global property of the game engine,
2. In the start() method, we added the event listeners for the keydown and keyup events, in each listener we will test if the arrow keys or the space bar has been pressed or released, and we set different properties of the inputStates object. For example if the spacebar is pressed, we do a inputStates.space = true; if it's released we do a nputStates.space = false.
3. In the mainLoop, we added some tests to chek what keys are down, if one key is down, we draw its name in the canvas.

Here is the online example you can try: http://jsbin.com/kafehi/9/edit



And here is the complete source code:

```
1  // Inits
2  window.onload = function init() {
3    var game = new GF();
4    game.start();
5  };
6
7
8  // GAME FRAMEWORK STARTS HERE
9  var GF = function(){
10     // Vars relative to the canvas
11     var canvas, ctx, w, h;
12
13     // vars for counting frames/s, used by the measureFPS function
14     var frameCount = 0;
15     var lastTime;
16     var fpsContainer;
17     var fps;
18
```

```
19      // vars for handling inputs
20      var inputStates = {};
21
22      var measureFPS = function(newTime){
23
24           // test for the very first invocation
25           if(lastTime === undefined) {
26             lastTime = newTime;
27             return;
28           }
29
30          //calculate the difference between last & current frame
31          var diffTime = newTime - lastTime;
32
33          if (diffTime >= 1000) {
34              fps = frameCount;
35              frameCount = 0;
36              lastTime = newTime;
37          }
38
39          //and display it in an element we appended to the
40          // document in the start() function
41         fpsContainer.innerHTML = 'FPS: ' + fps;
42         frameCount++;
43      };
44
45       // clears the canvas content
46       function clearCanvas() {
47         ctx.clearRect(0, 0, w, h);
48       }
49
50       // Functions for drawing the monster and maybe other objects
51       function drawMyMonster(x, y) {
52         // draw a big monster !
53         // head
54
55         // save the context
56         ctx.save();
57
58         // translate the coordinate system, draw relative to it
59         ctx.translate(x, y);
60
61         // (0, 0) is the top left corner of the monster.
62         ctx.strokeRect(0, 0, 100, 100);
63
64         // eyes
65         ctx.fillRect(20, 20, 10, 10);
66         ctx.fillRect(65, 20, 10, 10);
67
68         // nose
69         ctx.strokeRect(45, 40, 10, 40);
70
71         // mouth
72         ctx.strokeRect(35, 84, 30, 10);
73
74         // teeth
75         ctx.fillRect(38, 84, 10, 10);
76         ctx.fillRect(52, 84, 10, 10);
77
78        // restore the context
79        ctx.restore();
80      }
81
82      var mainLoop = function(time){
83          //main function, called each frame
84          measureFPS(time);
85
86          // Clear the canvas
87          clearCanvas();
88
89          // draw the monster
```

```
90          drawMyMonster(10+Math.random()*10, 10+Math.random()*10);
91          // check inputStates
92          if (inputStates.left) {
93              ctx.fillText("left", 150, 20);
94          }
95          if (inputStates.up) {
96              ctx.fillText("up", 150, 50);
97          }
98         if (inputStates.right) {
99              ctx.fillText("right", 150, 80);
100         }
101         if (inputStates.down) {
102             ctx.fillText("down", 150, 120);
103         }
104         if (inputStates.space) {
105             ctx.fillText("space bar", 140, 150);
106         }
107
108         // call the animation loop every 1/60th of second
109         requestAnimationFrame(mainLoop);
110     };
111
112     var start = function(){
113         // adds a div for displaying the fps value
114         fpsContainer = document.createElement('div');
115         document.body.appendChild(fpsContainer);
116
117         // Canvas, context etc.
118         canvas = document.querySelector("#myCanvas");
119
120         // often useful
121         w = canvas.width;
122         h = canvas.height;
123
124         // important, we will draw with this object
125         ctx = canvas.getContext('2d');
126         // default police for text
127         ctx.font="20px Arial";
128
129        //add the listener to the main, window object, and update the states
130       window.addEventListener('keydown', function(event){
131           if (event.keyCode === 37) {
132              inputStates.left = true;
133           } else if (event.keyCode === 38) {
134              inputStates.up = true;
135           } else if (event.keyCode === 39) {
136              inputStates.right = true;
137           } else if (event.keyCode === 40) {
138              inputStates.down = true;
139           }  else if (event.keyCode === 32) {
140              inputStates.space = true;
141           }
142       }, false);
143
144       //if the key will be released, change the states object
145       window.addEventListener('keyup', function(event){
146           if (event.keyCode === 37) {
147              inputStates.left = false;
148           } else if (event.keyCode === 38) {
149              inputStates.up = false;
150           } else if (event.keyCode === 39) {
151              inputStates.right = false;
152           } else if (event.keyCode === 40) {
153              inputStates.down = false;
154           } else if (event.keyCode === 32) {
155              inputStates.space = false;
156           }
157       }, false);
158
159
160         // start the animation
```

```
161         requestAnimationFrame(mainLoop);
162     };
163
164     //our GameFramework returns a public API visible from outside its scope
165     return {
166         start: start
167     };
168 };
169
```

You may notice that on some computers / Operating systems, it is not possible to press at the same time the up and down arrow keys, or left and right. They are mutually exclusive, however space + up + right is ok.

Dealing with mouse events

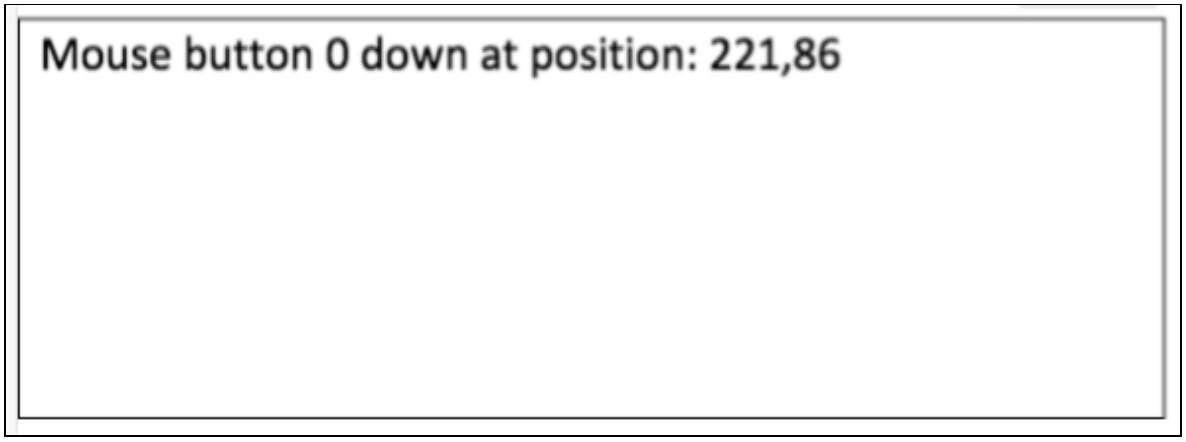
Working with mouse events means detect mouse button down/up, with identifying the button, keeping track of mouse moves, and also get the x, y coordinate of the cursor.

Special care must be taken when getting the mouse coordinates as the HTML5 canvas often has default CSS properties that would produce false coordinates. The trick to get the right x and y mouse cursor coordinate is to use this method from the canvas API:

```
1     // necessary to take into account CSS boudaries
2     var rect = canvas.getBoundingClientRect();
```

The width and the height of the rect object must be taken into account. They correspond to the padding / borders around the canvas. See how we deal with them in the getMousePos() function from the above example.

Here is an online example that covers all cases correctly: http://jsbin.com/bizudu/4/edit



Just move the mouse over the canvas, press or release mouse buttons. Notice that we keep the state of the mouse (position, buttons down or up) in the inputStates object, in a similar way we did with the keys in the previous section.

Here is the source code of this small test example:

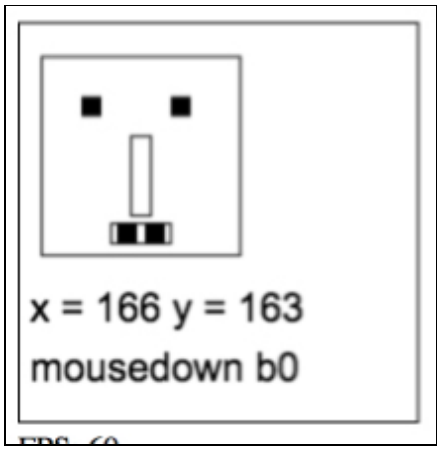
```
1  var canvas, ctx;
2  var inputStates = {};
3
4  window.onload = function init() {
5      canvas = document.getElementById('myCanvas');
6      ctx = canvas.getContext('2d');
7
8      canvas.addEventListener('mousemove', function (evt) {
9          inputStates.mousePos = getMousePos(canvas, evt);
```

```
10          var message = 'Mouse position: ' + inputStates.mousePos.x + ',' + inputStates.mouseP
11          writeMessage(canvas, message);
12      }, false);
13
14        canvas.addEventListener('mousedown', function (evt) {
15          inputStates.mousedown = true;
16          inputStates.mouseButton = evt.button;
17          var message = "Mouse button " + evt.button + " down at position: " + inputStates.mou
18          writeMessage(canvas, message);
19      }, false);
20
21          canvas.addEventListener('mouseup', function (evt) {
22          inputStates.mousedown = false;
23          var message = "Mouse up at position: " + inputStates.mousePos.x + ',' + inputStates.
24          writeMessage(canvas, message);
25      }, false);
26  };
27
28  function writeMessage(canvas, message) {
29      var ctx = canvas.getContext('2d');
30      ctx.save();
31      ctx.clearRect(0, 0, canvas.width, canvas.height);
32      ctx.font = '18pt Calibri';
33      ctx.fillStyle = 'black';
34      ctx.fillText(message, 10, 25);
35      ctx.restore();
36  }
37
38  function getMousePos(canvas, evt) {
39      // necessary to take into account CSS boudaries
40      var rect = canvas.getBoundingClientRect();
41      return {
42          x: evt.clientX - rect.left,
43          y: evt.clientY - rect.top
44      };
45  }
46
47
```

Include the mouse listeners into the game engine

Now we will include these listeners into our game engine. Notice that we changed some parameters (no need to pass the canvas as a parameter of the getMousePos function, for example).

The new online version of the game engine can be tried at: http://jsbin.com/kafehi/12/edit



Complete source code:

```
1   // Inits
2   window.onload = function init() {
```

```
  var game = new GF();
  game.start();
};


// GAME FRAMEWORK STARTS HERE
var GF = function(){
    // Vars relative to the canvas
    var canvas, ctx, w, h;

    // vars for counting frames/s, used by the measureFPS function
    var frameCount = 0;
    var lastTime;
    var fpsContainer;
    var fps;

    // vars for handling inputs
    var inputStates = {};

    var measureFPS = function(newTime){

         // test for the very first invocation
         if(lastTime === undefined) {
           lastTime = newTime;
           return;
         }

        //calculate the difference between last & current frame
        var diffTime = newTime - lastTime;

        if (diffTime >= 1000) {
            fps = frameCount;
            frameCount = 0;
            lastTime = newTime;
        }

        //and display it in an element we appended to the
        // document in the start() function
       fpsContainer.innerHTML = 'FPS: ' + fps;
       frameCount++;
    };

     // clears the canvas content
     function clearCanvas() {
       ctx.clearRect(0, 0, w, h);
     }

     // Functions for drawing the monster and maybe other objects
     function drawMyMonster(x, y) {
       // draw a big monster !
       // head

       // save the context
       ctx.save();

       // translate the coordinate system, draw relative to it
       ctx.translate(x, y);

       // (0, 0) is the top left corner of the monster.
       ctx.strokeRect(0, 0, 100, 100);

       // eyes
       ctx.fillRect(20, 20, 10, 10);
       ctx.fillRect(65, 20, 10, 10);

       // nose
       ctx.strokeRect(45, 40, 10, 40);

       // mouth
       ctx.strokeRect(35, 84, 30, 10);

```

```
74          // teeth
75          ctx.fillRect(38, 84, 10, 10);
76          ctx.fillRect(52, 84, 10, 10);
77
78        // restore the context
79        ctx.restore();
80      }
81
82      var mainLoop = function(time){
83          //main function, called each frame
84          measureFPS(time);
85
86          // Clear the canvas
87          clearCanvas();
88
89          // draw the monster
90          drawMyMonster(10+Math.random()*10, 10+Math.random()*10);
91          // check inputStates
92          if (inputStates.left) {
93              ctx.fillText("left", 150, 20);
94          }
95          if (inputStates.up) {
96              ctx.fillText("up", 150, 40);
97          }
98         if (inputStates.right) {
99              ctx.fillText("right", 150, 60);
100         }
101         if (inputStates.down) {
102             ctx.fillText("down", 150, 80);
103         }
104         if (inputStates.space) {
105             ctx.fillText("space bar", 140, 100);
106         }
107         if (inputStates.mousePos) {
108             ctx.fillText("x = " + inputStates.mousePos.x + " y = " + inputStates.mousePos.y
109         }
110       if (inputStates.mousedown) {
111             ctx.fillText("mousedown b" + inputStates.mouseButton, 5, 180);
112         }
113
114         // call the animation loop every 1/60th of second
115         requestAnimationFrame(mainLoop);
116     };
117
118
119     function getMousePos(evt) {
120         // necessary to take into account CSS boudaries
121         var rect = canvas.getBoundingClientRect();
122         return {
123             x: evt.clientX - rect.left,
124             y: evt.clientY - rect.top
125         };
126     }
127
128     var start = function(){
129         // adds a div for displaying the fps value
130         fpsContainer = document.createElement('div');
131         document.body.appendChild(fpsContainer);
132
133         // Canvas, context etc.
134         canvas = document.querySelector("#myCanvas");
135
136         // often useful
137         w = canvas.width;
138         h = canvas.height;
139
140         // important, we will draw with this object
141         ctx = canvas.getContext('2d');
142         // default police for text
143         ctx.font="20px Arial";
144
```

```
145         //add the listener to the main, window object, and update the states
146         window.addEventListener('keydown', function(event){
147             if (event.keyCode === 37) {
148                 inputStates.left = true;
149             } else if (event.keyCode === 38) {
150                 inputStates.up = true;
151             } else if (event.keyCode === 39) {
152                 inputStates.right = true;
153             } else if (event.keyCode === 40) {
154                 inputStates.down = true;
155             }  else if (event.keyCode === 32) {
156                 inputStates.space = true;
157             }
158         }, false);
159 
160         //if the key will be released, change the states object
161         window.addEventListener('keyup', function(event){
162             if (event.keyCode === 37) {
163                 inputStates.left = false;
164             } else if (event.keyCode === 38) {
165                 inputStates.up = false;
166             } else if (event.keyCode === 39) {
167                 inputStates.right = false;
168             } else if (event.keyCode === 40) {
169                 inputStates.down = false;
170             } else if (event.keyCode === 32) {
171                 inputStates.space = false;
172             }
173         }, false);
174 
175         // Mouse event listeners
176         canvas.addEventListener('mousemove', function (evt) {
177             inputStates.mousePos = getMousePos(evt);
178         }, false);
179 
180         canvas.addEventListener('mousedown', function (evt) {
181               inputStates.mousedown = true;
182               inputStates.mouseButton = evt.button;
183         }, false);
184 
185         canvas.addEventListener('mouseup', function (evt) {
186             inputStates.mousedown = false;
187         }, false);
188 
189 
190           // start the animation
191           requestAnimationFrame(mainLoop);
192       };
193 
194       //our GameFramework returns a public API visible from outside its scope
195       return {
196           start: start
197       };
198   };
```

Making the monster move using the arrow keys, increase its speed when pressing a mouse button

To conclude this section, we will now use the arrow keys to move the monster up/down/left/right from the previous examples, and make it speed up when we press a mouse button while it moves. Notice that pressing two keys at the same time make it move diagonally.

Check this online example, we changed only a few lines of code from the previous one:
http://jsbin.com/bemebi/2/edit

We first added a variable for describing the monster :

```
// The monster !
var monster = {
  x:10,
  y:10,
  speed:1
};

```

Where monster.x and monster.y will define the current monster position, and monster.speed corresponds to the number of pixels we will move the monster vertically or horizontally between each frames of animation (when an arrow key is pressed). Note: this is not the best way to animate objects in a game, we will look at a much proper solution the next week, named "time based animation".

We modified the game loop this way:

```
var mainLoop = function(time){
    //main function, called each frame
    measureFPS(time);

    // Clear the canvas
    clearCanvas();

    // draw the monster
    drawMyMonster(monster.x, monster.y);

    // Check inputs and move the monster
    updateMonsterPosition();

    // call the animation loop every 1/60th of second
    requestAnimationFrame(mainLoop);
};
```

We moved all the parts that checks the input states in the updateMonsterPosition() function:

```
function updateMonsterPosition() {
  monster.speedX = monster.speedY = 0;
    // check inputStates
    if (inputStates.left) {
        ctx.fillText("left", 150, 20);
        monster.speedX = -monster.speed;
    }
    if (inputStates.up) {
        ctx.fillText("up", 150, 40);
       monster.speedY = -monster.speed;
    }
   if (inputStates.right) {
        ctx.fillText("right", 150, 60);
        monster.speedX = monster.speed;
    }
    if (inputStates.down) {
        ctx.fillText("down", 150, 80);
        monster.speedY = monster.speed;
    }
    if (inputStates.space) {
        ctx.fillText("space bar", 140, 100);
    }
    if (inputStates.mousePos) {
        ctx.fillText("x = " + inputStates.mousePos.x + " y = " + inputStates.mousePos.y,
    }
   if (inputStates.mousedown) {
        ctx.fillText("mousedown b" + inputStates.mouseButton, 5, 180);
        monster.speed = 5;
    } else {
      // mouse up
      monster.speed = 1;
```

```
32          }
33
34          monster.x += monster.speedX;
35          monster.y += monster.speedY;
36
37      }
```

In this function we added on the fly two properties to the monster object: speedX and speedY that will correspond to the number of pixels we will add to the x and y position of the monster.

We first set these to zero (line 2), then depending on the keyboard input states, we set them to a value equal to monster.speed or -monster.speed depending on the keys that are being pressed (lines 4-20).

Finally, we add speedX and speedY pixels to the x and y position of the monster (line 35 and 36). As the function is called by the game loop, of speedX or speedY are different from zero, this will change the x and y position of the monster every frame, making it moving smoothly.

In case a mouse button is pressed or released we set the monster.speed value to +5 or to +1. This will make the monster go faster when a mouse button is down, go back to its normal speed when no button is down.

Notice that two arrow keys can be pressed at once + the mouse down at the same time, in that case the monster will take a diagonal direction + speed up. This is why we had to keep all the input states up to date, and not handle single key events.

7 What's next? What is missing?

Wow, we just introduced basic concepts... many things need to be seen like:

- Look at what we can draw in a canvas: shapes, images, etc.
- Time based animation
- Look at some specific drawing techniques like animating sprites (image based animation), or vectorial drawing,
- Collision detection,
- Sound effects and music,
- Game states (splash screen, welcome menu, game over, etc..)
- Persistence (save high scores)
- Asynchronous loading of resources at the beginning of the game (load images, sprite sheets, sounds, etc)
- How to make a networked game that be played in real time by several players...
- Etc...

This will give us work for the next weeks :-)