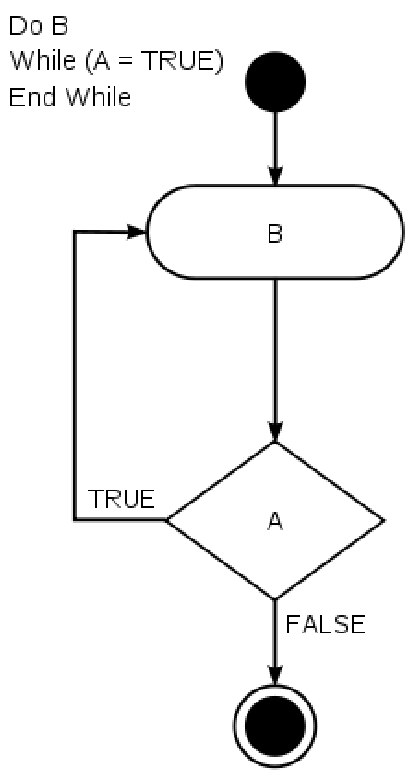
The "game loop"

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 processors format text as a user types. If the user doesn't type anything, the word processor is waiting for an action.

Games operate differently: a game must continue to operate regardless of a user's input!

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

DIFFERENT IMPLEMENTATIONS OF THE 'MAIN GAME LOOP'

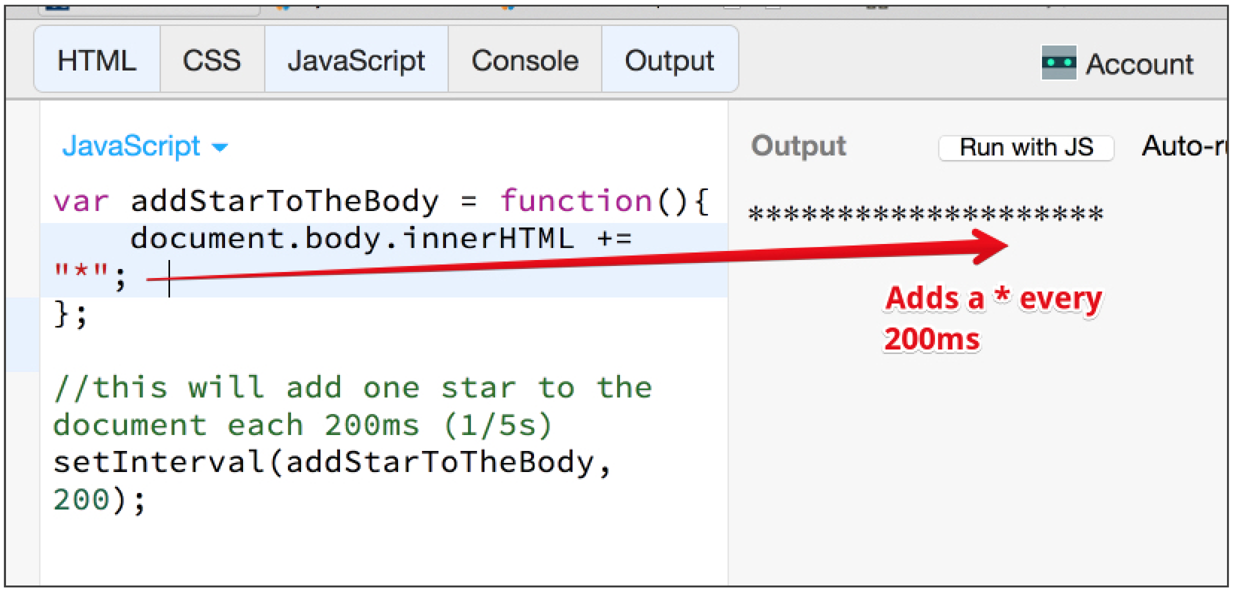
There are different ways to perform animation with JavaScript. A very detailed comparison of three different methods has already been presented in the HTML5 Part 1 course, week 4. Below is a quick reminder of the methods, illustrated with new, short online examples.

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 this by calling the `clearInterval(id)` function with the interval identifier as an argument.

Try an example at JSBin : open the HTML, JavaScript and output tabs to see the code.



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

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

```
********************
```

Source code extract:

```
var addStarToTheBody = function(){
   document.body.innerHTML += "*";
};
//this will add one star to the document each 200ms (1/5s)
setInterval(addStarToTheBody, 200);
```

WRONG:

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

GOOD:

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

or like we did in the example, with an external function.

Using `setTimeout()` instead of `setInterval()`

Reminder from HTML5 part 1 course, week 4: with `setInterval` - if we set the number of milliseconds at, say, 200, it will call our game loop function EACH 200ms, even if the previous one is not yet finished.

So instead, we can use another function, better suited to our goals.

- Syntax: `setTimeout(function, ms);`

This function works like `setInterval` with one little difference: it calls your function AFTER a given amount of time.

Try an example at JSBin: open the HTML, JavaScript and output tabs to see the code. This example does the same thing as the previous example: adds a "*" to the document each 200ms.

Source code extract:

```
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 the previous example. It is far better, however, because the timer waits for the function to finish everything inside before calling it back again.

For several years, `setTimeout` was the best and most popular JavaScript implementation of game loops. This changed when Mozilla presented the requestAnimationFrame API, which became the reference W3C standard API for game animation.

Using the `requestAnimationFrame` API

- Note: how to use `requestAnimationFrame` has been covered in detail in the week 4 of the HTML5 Part 1 course.

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 or a canvas during every loop? Or maybe you just want to make some calculations or send requests a couple of times per second? For this reason, 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 introduced a new approach which they called `requestAnimationFrame`.

This approach helps your browser to optimize all the animations on the screen, no matter whether 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 JSBin.

Source code extract:

```
window.onload = function init() {
   // called after the page is entirely loaded
   requestAnimationFrame(mainloop);
};
function mainloop(timestamp) {
   document.body.innerHTML += "*";
   // call back itself every 60th of second
   requestAnimationFrame(mainloop);
```

```
    }
```

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 of a second.

This target may be hard to reach; the animation loop content may take longer than this, or the scheduler may be a bit late or early.

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 between each animation frame. 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 useful for exactly that: it gives a high resolution time.

Polyfills for requestAnimationFrame

Note that "old browsers" have implemented some prefixed, experimental versions of the API. In some tutorials found on the Web, you might encounter a piece of code that uses`requestAnimationFrame` with a polyfill that will ensure that the examples work on any browser, including those that do not support this API at all (falling back to `setTimeout`). This is mainly the case with old Internet Explorers (versions 6-9).

The most famous shim has been written by Paul Irish from the jQuery team. He wrote this shim to simplify the usage of`requestAnimationframe` in different browsers (look at WikiPedia for the meaning of "shim").

```
// 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);
     };
})();
```

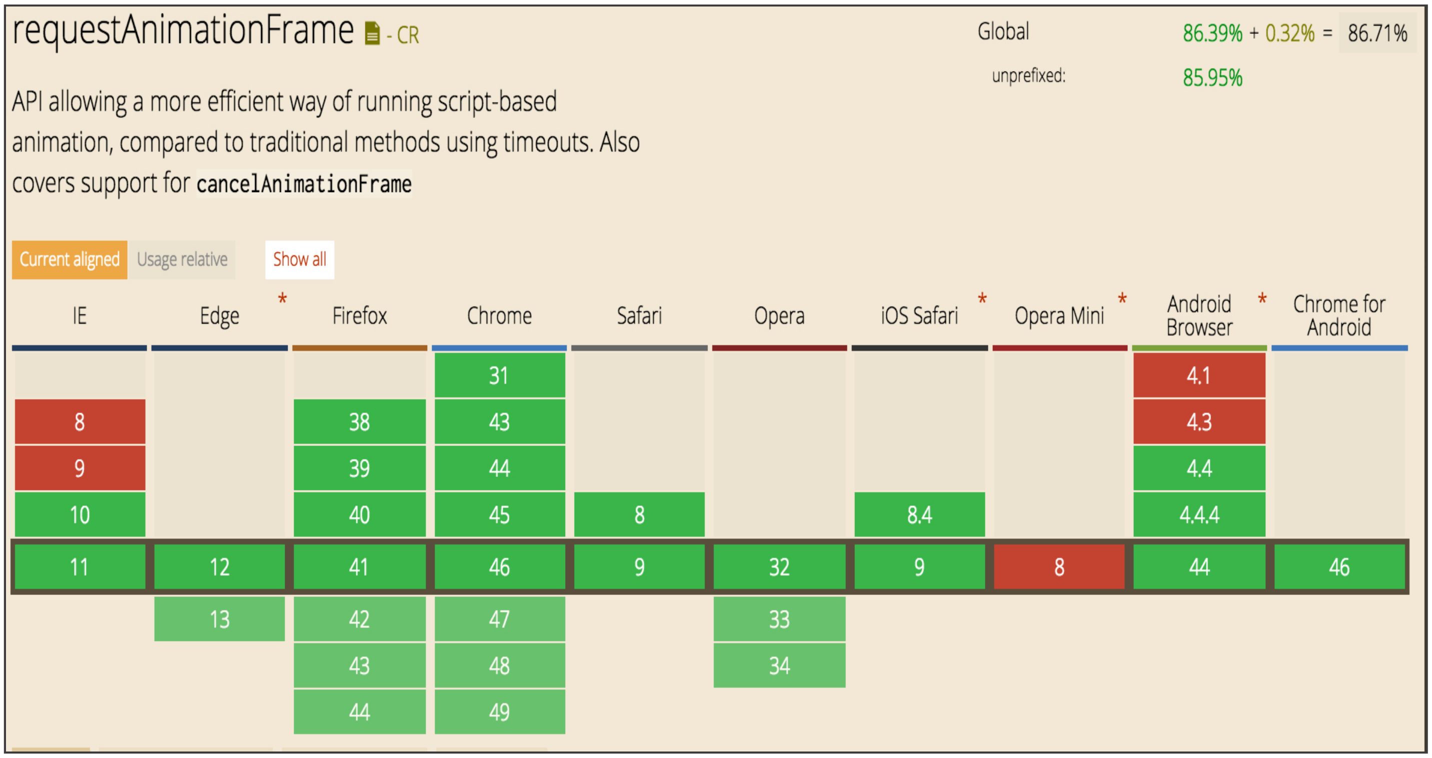
So according to our last example, using `requestAnimationFrame`with this shim will look like this :

```
// 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 shim defines a function named `requestAnimFrame`(instead of the standard `requestAnimationFrame`).

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

Current support:



requestAnimationFrame - CR

Global 86.39% + 0.32% = 86.71%

unprefixed: 85.95%

API allowing a more efficient way of running script-based animation, compared to traditional methods using timeouts. Also covers support for `cancelAnimationFrame`

Current aligned | Usage relative | Show all

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
			31					4.1	
8		38	43					4.3	
9		39	44					4.4	
10		40	45	8		8.4		4.4.4	
11	12	41	46	9	32	9	8	44	46
	13	42	47		33				
		43	48		34				
		44	49						

Up to date version of this table.