Animating with javascript: from setInterval to requestAnimationFrame

on August 3, 2011 by louisremi
in Animations DOM JavaScript

- 12 comments
- Share This

Animating DOM elements[1] or the content of a canvas is a classical use case for setInterval. But the interval is not as reliable as it seems, and a more suitable API is now available…

Animating with setInterval

To animate an element moving 400 pixels on the right with javascript, the basic thing to do is to move it 10 pixels at a time on a regular interval.
JSFiddle demo.
An HTML5 game based on this logic would normally run at ~60fps[2], but if the animations were too complex or running on a low-spec. device (a mobile phone for instance) and processing a frame were taking more than 16ms, then the game would run at a lower framerate: when processing 1 frame takes 33ms, the game runs at 30fps and game elements move twice as slowly as they should. Animations would still look smooth enough, but the game experience would be altered.

Animating at constant speed

To animate at constant speed, we need to calculate the time delta since the last frame and move the element proportionally.

```
var elem = document.getElementById("animatedElem"),
  left = 0,
  lastFrame = +new Date,
  timer;
// Move the element on the right at ~600px/s
timer = setInterval(function() {
  var now = +new Date,
    deltaT = now - lastFrame;
  elem.style.left = ( left += 10 * deltaT / 16 ) + "px";
  lastFrame = now;
  // clear the timer at 400px to stop the animation
  if ( left > 400 ) {
    clearInterval( timer );
  }
}, 16);
```

view raw **intervalLoop_constant.js** hosted with ❤ by **GitHub**

Animating with requestAnimationFrame

Since the interval parameter is irrelevant in complex animations, as there's no guarantee that it will be honored, a new API has been designed:requestAnimationFrame. It's simply a way to tell the browser "before drawing the next frame on the screen, execute this game logic/animation processing". The browser is in charge of choosing the best moment to execute the code, which results in a more efficient use of resources[3].
Here's how an animation with requestAnimationFrame would be written.
Note: Following code snippets don't include feature detections and workarounds necessary to work in current browsers. Should you want to play with them, you should try the ready-to-use animLoop.js.

```
function animLoop( render, element ) {
    var running, lastFrame = +new Date;
    function loop( now ) {
        // stop the loop if render returned false
        if ( running !== false ) {
            requestAnimationFrame( loop, element );
```

```
            running = render( now - lastFrame );
            lastFrame = now;
        }
    }
    loop( lastFrame );
}

// Usage
animLoop(function( deltaT ) {
    elem.style.left = ( left += 10 * deltaT / 16 ) +
"px";
    if ( left > 400 ) {
      return false;
    }
// optional 2nd arg: elem containing the animation
}, animWrapper );
```

view rawanimLoop_raf.js hosted with ♥ by GitHub

Dealing with inactive tabs

requestAnimationFrame was built with another benefit in mind: letting the browser choose the best frame interval allows to have a long interval in inactive tabs. Users could play a CPU intensive game, then open a new tab or minimize the window, and the game would pause[4], leaving resources available for other tasks.

Note: the potential impact of such behavior on resource and battery usage is so positive that browser vendors decided to adopt it for setTimeout and setInterval as well[5].

This behavior also means that the calculated time delta might be really high when switching back to a tab containing an animation. This will result in animation appearing to jump or creating "wormholes"[6], as illustrated here.

Wormholes can be fixed by clamping the time delta to a maximum value, or not rendering a frame when the time delta is too high.

```
function animLoop( render, element ) {
    var running, lastFrame = +new Date;
    function loop( now ) {
        // stop the loop if render returned false
        if ( running !== false ) {
            requestAnimationFrame( loop, element );
            var deltaT = now - lastFrame;
            // do not render frame when deltaT is too high
            if ( deltaT < 160 ) {
                running = render( deltaT );
            }
            lastFrame = now;
        }
    }
    loop( lastFrame );
}
```

12345678910111213141516

view rawanimLoop_fixed.js hosted with ♥ by GitHub

JSFiddle demo.

Problems with animation queues

Libraries such as jQuery queue animations on elements to execute them one after the other. This queue is generally only used for animations that are purposefully consecutive.

But if animations are triggered by a timer, the queue might grow without bound in inactive tabs, as paused animations stack up in the queue. When switching back to affected tabs, a user will see a large number of animations playing consecutively when only one should happen on a regular interval:

JSFiddle demo.

This problem is visible in some auto-playing slideshows such as mb.gallery. To work around it, developers can empty animation queues before triggering new animations[7].

JSFiddle demo.

Conclusion

The delays of setTimeout and setInterval and of course requestAnimationFrame are unpredictable and much longer in inactive tabs. These facts should be taken into account not only when writing

animation logic, but in fps counters, time countdowns, and everywhere time measurement is crucial.

[1] The DOM can now be animated with CSS3 Transitions and CSS3 Animations.

[2] 1 frame every 16ms is 62.5 frames per second.

[3] See the illustration of this fact on msdn.

[4] The behavior of requestAnimationFrame in inactive tabs is still being worked on at the w3c, and might differ in other browsers.

[5] See related Firefox bug and related chromium bug.

[6] This term was first coined by Seth Ladd in his “Intro to HTML5 Game Development” talk.

[7] See documentation of your js library, such as effects and stop() for jQuery.