ANIMATIONS ON FIRE

Brian Birtles, Mozilla Japan

Graphical Web 2014, Winchester

HTML version of slides: http://people.mozilla.org/~bbirtles/pres/graphical-web-2014/
ANIMATION IS AWESOME...

Animations can be used for more than just cat gifs. They can be used to tell stories too.
Animation is essentially about using time to convey information.

Animation can be used as component of user interface design to describe the results of an action.
It can be more intuitive without cluttering the screen or requiring extra explanation.
But when animation runs slowly or hesitates, that information is lost. Hence for animation, performance is critical.

In order to fix animation performance in Web pages, we really need to understand how browsers work.
As we follow the journey from markup to our eyeballs, we will consider how we can make each step smoother or skip it all together.
Parsing can be slow. Most browsers do it in a separate thread. If we skip parsing while animating surely it goes faster?
## setAttribute VS SVG DOM

<table>
<thead>
<tr>
<th>Browser</th>
<th><code>setAttribute('cx', 'X')</code></th>
<th><code>cx.baseVal.value = X</code></th>
<th>% improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox 34</td>
<td>246.6</td>
<td>131.4</td>
<td>47%</td>
</tr>
<tr>
<td>Chrome 36</td>
<td>155.8</td>
<td>20.4</td>
<td>87%</td>
</tr>
<tr>
<td>IE 11</td>
<td>2347.6</td>
<td>1897.4</td>
<td>19%</td>
</tr>
</tbody>
</table>

### transform

<table>
<thead>
<tr>
<th>Browser</th>
<th><code>setAttribute('transform', 'translate(X, Y)')</code></th>
<th><code>transform.baseVal[0].matrix.{e,f} = {X,Y}</code></th>
<th>% improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox 34</td>
<td>258.4</td>
<td>224.8</td>
<td>13%</td>
</tr>
<tr>
<td>Chrome 36</td>
<td>199.6</td>
<td>30.6</td>
<td>85%</td>
</tr>
<tr>
<td>IE 11</td>
<td>1922.8</td>
<td>2592</td>
<td>-35%</td>
</tr>
</tbody>
</table>

* Times are ms taken for 100,000 iterations averaged over 5 runs (lower numbers are faster)

A micro-benchmark suggests an API that skips parsing is faster.
How about in a real-world animation?
It doesn’t make a lot of difference. Perhaps 3~4 fps at best.
More realistic test

Try it at home!
Try using a specialized API to avoid parsing
There are bigger performance gains to be had from the style system.
What happens if we exploit the fact that display: none elements don’t appear in the render tree?

define the DOM (content) tree:
- Document
  - html
    - head
    - body
      - figure
        - img
        - figcaption
          - "My cat"

define the render (frame) tree:
- Viewport (html)
  - Scroll (html)
  - Block (html)
  - Block (body)
    - Inline (figure)
      - Image (img)
      - Block (figcaption)
      - Text

style definition:

```css
figure {
  display: inline;
}
figcaption {
  display: none;
}
```

* Shaded nodes represent the tree if the `<figcaption>` was display: inline-block.
## USING `display:none`

<table>
<thead>
<tr>
<th>Browser</th>
<th>Unoptimized</th>
<th>With <code>display:none</code></th>
<th>Avg. improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox 34</td>
<td>25.6fps</td>
<td>29fps</td>
<td>3.4fps / 13%</td>
</tr>
<tr>
<td>Chrome 36</td>
<td>5.1fps</td>
<td>12.5fps</td>
<td>7.4fps / 145%</td>
</tr>
<tr>
<td>IE 11</td>
<td>4.8fps</td>
<td>7.8fps</td>
<td>3.0fps / 63%</td>
</tr>
</tbody>
</table>

*Average result after 3–5 runs. Higher numbers are better.*

*Firefox doesn’t show such a big difference in this case since the test case animates ‘top’ which, as we’ll see, does not trigger reflow in Firefox so setting `display:none` doesn’t have as big an impact.*

---

**Try it at home!**

Remove elements from the render tree with `display:none`
This technique improved performance for the Parapara animation project where characters are set to display:none when they are off-stage.
Of the operations performed in the style system, the **layout/reflow** step is often expensive.
We can measure style resolution and layout time in profiling tools in Firefox (above) and Chrome (below).
Firefox lets you inspect reflow (layout) in the console.
Let's see how different animations affect layout

- Animating margin-left
- Animating left (position: relative)
- Animating transform
Animating `margin-left`

Animating `left` (position: relative)

Animating margin-top causes reflow on every frame
But in Firefox, animating top or transform does not trigger reflow (layout)
# TRIGGERING LAYOUT

<table>
<thead>
<tr>
<th>Property</th>
<th>Firefox 34</th>
<th>Chrome 36</th>
<th>IE 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>margin-left</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>left</code> (position: relative)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>left</code> (position: absolute)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>transform</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Based on my inspection of profiles from this test case.
Comparing the performance of margin-top and transform, transform is faster because it avoids reflow but it also benefits from layerization which we will see later.
AVOIDING REFLOW

Try `transform` instead of `top / left / marginTop` etc.

Try animating elements that are `position:absolute`.

Try non-geometric properties like `color, opacity` etc.

Try `transform: scale` instead of `font-size`.
Since these processes can be expensive, browsers avoid doing them until either they have to paint, or until script asks a question about the current state.
WHAT TRIGGERS RECALC / REFLOW?

- `window.getComputedStyle(elem).color`
  → style recalc (typically)

- `window.getComputedStyle(elem).width`,
  `elem.offsetTop`, `elem.getClientRects()`, etc.

  → reflow
DON'T DO THIS

```javascript
for (var i = 1; i < containerElem.children.length; i++) {
    containerElem.children[i].style.top =
    containerElem.children[i-1].offsetTop + 10 + "px";
}
```
# Avoiding Forcing Reflow

<table>
<thead>
<tr>
<th>Browser</th>
<th>Triggering reflow</th>
<th>Not doing that</th>
<th>Avg. improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox 34</td>
<td>42.1fps*</td>
<td>45.8fps*</td>
<td>3.7fps / 9%</td>
</tr>
<tr>
<td>Chrome 36</td>
<td>10.5fps</td>
<td>23.2fps</td>
<td>12.7fps / 120%</td>
</tr>
<tr>
<td>IE 11</td>
<td>8.2fps</td>
<td>19.1fps</td>
<td>10.9fps / 132%</td>
</tr>
</tbody>
</table>

* Average result after 3–5 runs of test A and test B. Results for Firefox were particularly variable but were generally only slightly faster since the test animates the `top` property which does not trigger reflow in Firefox.

---

**Try it at home!**

Try reading computed style (especially geometry) less often or not at all.
Painting is often the most expensive part. Firefox creates a display list of items to paint, then creates a layer tree into which it paints. The layers in the tree are then composited together.
PAINT COST

Box #1

Box #2

Paint area

×

Paint complexity

svg

WOW!
We can see exactly what area is being painted

**PAINT FLASHING (FIREFOX)**
PAINT RECTANGLES (CHROME)

- Capture stacks
- Capture memory

Records

- Show paint rectangles
- Show composited layer borders
- Show FPS meter
- Enable continuous page repainting
- Show potential scroll bottlenecks
PAINT FLASHING #1
When animating transform we only paint once at the start. This is because it gets put in its own layer.
When animating independent areas Chrome seems to paint the union of dirty areas so layerization can be more important there.
However, SVG filters are often hardware accelerated. Sometimes the combination of features is what is slow.
Try it at home!

Try replacing expensive effects with simpler ones.
PRE-RENDERING

We can sometimes make things faster by pre-rendering. Desktop apps, native apps, Flash apps, everyone does it.

<table>
<thead>
<tr>
<th>Browser</th>
<th>&lt;iframe src=&quot;svg&quot;&gt;</th>
<th>&lt;img src=&quot;png&quot;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox 34</td>
<td>1.9 fps</td>
<td>49.5 fps</td>
</tr>
<tr>
<td>Chrome 36</td>
<td>11.18 fps</td>
<td>49.7 fps</td>
</tr>
<tr>
<td>IE 11</td>
<td>5.8 fps</td>
<td>50.9 fps</td>
</tr>
</tbody>
</table>

Try it at home! Pre-render expensive assets
# Pre-Rendering

<table>
<thead>
<tr>
<th>Browser</th>
<th><code>&lt;iframe src=&quot;svg&quot;&gt;</code></th>
<th><code>&lt;img src=&quot;svg&quot;&gt;</code></th>
<th><code>&lt;img src=&quot;png&quot;&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox 34</td>
<td>1.9 fps</td>
<td>49.1 fps</td>
<td>49.5 fps</td>
</tr>
<tr>
<td>Chrome 36</td>
<td>11.18 fps</td>
<td>13.0 fps*</td>
<td>49.7 fps</td>
</tr>
<tr>
<td>IE 11</td>
<td>5.8 fps</td>
<td>15.5 fps</td>
<td>50.9 fps</td>
</tr>
</tbody>
</table>

* Some rendering defects

**Try it at home!**

Try using `<img>` to embed SVG images instead of `<iframe>` (or `<object>`, `<embed>`).

Alternatively, for SVG, simply referring to the SVG using `<img>` instead of `<iframe>` lets the browser make more optimizations. Especially Firefox.
Most browsers hardware accelerate layer compositing. That means they can often paint an animated element once then just change its transform, opacity etc. and let the GPU re-compose. That saves a lot of painting.
WHAT GETS A LAYER?

It’s up to the browser what gets a layer. Typically it’s things like the above.
INSPECTING LAYERS

• Firefox: `about:config`
  → `layers.draw-borders` to true
  (requires `layers.offmainthreadcomposition.enabled` to be true)

• Chrome: DevConsole → Rendering → Show composited layer borders
INSPECTING LAYERS

Animated transform

Animated opacity

3D transform
INSPECTING LAYERS

Animated transform

Animated opacity

3D transform
In the previous example, we can see why the transform animation only gets painted once. That element has its own layer.
Layerization is performed by the browser so it can automatically do it for SVG (SMIL) animation too.
And even for scripted animation, the browser can detect that an element is moving a lot and decide it would benefit from being on a separate layer.

(The red boxes in this example indicate image layers.)
Often, however, the browser won’t create a layer until an element starts animating. Sometimes that can be too late and can cause the animation to stutter at the start as the browser sets up the layer.
will-change:<property>
- will-change:transform
- will-change:opacity

will-change:scroll-position
will-change:contents

transform:translateZ(0)

Firefox: need layout.css.will-change.enabled in about:config.
APPLYING will-change

Apply will-change
APPLYING will-change

Apply will-change
Apart from low frame-rates, animation performance is affected by other processes on the same thread like layout, garbage collection, or other scripts, that cause the animation to stop and start (jank).
To avoid jank, some animations can be run on a separate thread/process.
Unlike animations running on the main thread which stop and start...
ANIMATION ON THE COMPOSITOR

Spin with script  Spin with CSS

Interrupt

... these animations continue along uninterrupted.
But not everything can be animated in this way. In particular, when the browser doesn’t know all the parameters of the animation—like most scripted animations—the browser can’t delegate the animation to another thread/process.

NOT SO FAST...

- Representable by compositors? (e.g. transform, opacity)
- Supported platform? (e.g. Firefox OS)
  (layers.offmainthreadcomposition.async-animations → true)
- Other limitations: top/left also animated?
- Controlled by the browser? (e.g. CSS Animations)
One way around this is to use the Web Animations API to create animations. This lets the browser optimize the animation in the same way as it does for declarative animations like CSS Animations/Transitions.

```javascript
elem.animate({
  transform: 'rotate(360deg)',
  duration: 1200,
  iterations: Infinity
});
```

- Chrome 36
- Polyfill `web-animations-js` and `web-animations-next`
Summarizing our journey...
Parse → DOM Tree → Render Tree

(function() {
    elem.remove();
};

article {
    margin: 1em;
}

Try using typed API to avoid parsing.
Using our knowledge of how browsers work we can make animations that run smoothly on any browser on any device and convey their intended effect.
Web Animations spec
dev.w3.org/fxtf/web-animations/

Brian Birtles
bbirtles@mozilla.com
@brianskold

HTML version of slides: http://people.mozilla.org/~bbirtles/pres/graphical-web-2014/