

Chrome: Plugin Power Saver

Status: Draft

Authors: tommycli@chromium.org, grobby@chromium.org, dbeam@chromium.org

Last Updated: 2015-06-29

Objective

Background

Overview

Detailed Design

Non-Essential Plugin Detection

Preview Keyframe Extraction

Poster Images: Explicitly Specified Preview Images

Preroll Keyframe Extraction

Plugin Pausing and Resuming

Throttling using Fake Offscreen Technique

Rollout

Project Information

Security Considerations

Power Benchmarks

Objective

1. Reduce Chrome power consumption on Chromebooks and laptops.
2. Make plugin media heavy websites more responsive.

Background

Chrome consumes a lot of power while performing heavy workloads. See [Power Benchmarks](#).

Safari 7 replaces non-essential plugin animations with a static image preview of the animation. To the user, the Flash content simply looks “paused”. When the user clicks on the “paused” Flash content, the animation resumes.

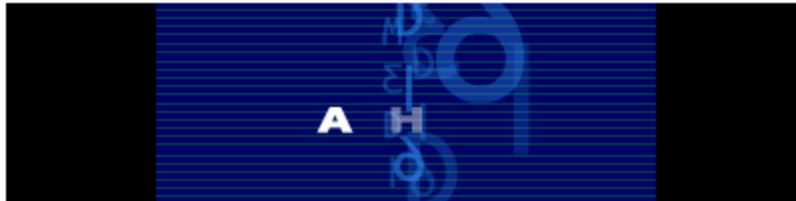
Safari still runs plugin content that’s determined to be the “main attraction” of the page. This project doesn’t address potential performance differences in that case.

Overview

Chrome should replace non-essential plugin content with a static image preview. To the user, the plugin should simply look “paused”. See demo video:

http://youtu.be/cX_ssTvMaq8

Plugin in active state:



Plugin in paused state (with UI overlay):



Once the user clicks on the “paused” plugin, the plugin should appear to resume. The click is not sent through to the plugin. For example, if clicking on the plugin would have resulted in a navigation, the click will not be a clickthrough.

Additionally, for background tabs that have never been foregrounded, such as tabs from session restore or middle-click open, plugins are deferred until the tab is foregrounded for the first time.

Detailed Design

We are adding a DETECT (“Detect and run important content”) option to the Content Settings for Plugins. We are migrating all existing ASK (“Click to Play”) users to BLOCK (“Let me choose when to run plugins”). DETECT will become the default option as early as Chrome 45.

A successful implementation will have these components.

1. **Non-Essential Plugin Detection** - We don’t want to throttle plugins that are in the “main attraction” of a page. That would make a website broken.
2. **Preview Keyframe Extraction** - We need a representative “preview image” which will act as a placeholder for the user to click to play the plugin.
3. **Plugin Pausing and Resuming** - A paused plugin should not take system resources. A

resumed plugin should not break scripting.

Non-Essential Plugin Detection

We should throttle plugin content that the user likely does not want to interact with. We should leave plugin content in the “main attraction” of a page running at full speed.

We divide plugin content into *essential* content and *peripheral* content. Only *peripheral* content is throttled.

We are using a heuristic based on size and plugin site origin:

- All same-origin (as the top level frame) plugin content is *essential*.
- All tiny content (smaller than 5x5) is *essential*.
- Cross-origin plugin content is *peripheral* if all plugin content from that origin is *small*.
 - *Small* content is smaller than 398px in width or smaller than 298px in height.
 - For instance, 300x200, 399x600, and 600x299 are all *small*.
- **Heuristics are subject to change.** Refer to the actual source code [here](#).

We took guidance from heuristics other browsers use, as we want to minimize surprises and incompatibilities for web developers.

Preview Keyframe Extraction

A good preview image meets these criteria:

1. **Makes the website look correct** - This feature can't make websites look broken or worse than if the plugin was still animating.
2. **Informs the user of the plugin content** - The user needs to know what to expect if they click on the plugin preview to resume the animation.
3. **Can be retrieved quickly**
 - a. Ideally, we want to immediately display an appropriate static preview.
 - b. If we must run the plugin, we want animation to stop ASAP. Another option is to preroll the plugin invisibly. This can make the page appear to load slowly, however.
 - c. Our technique must be resource lightweight, as this loads the system at the worst possible time (page load).

There are two types of of preview keyframes we can use:

1. Explicitly Specified Preview Images
2. Heuristic-based Keyframe Extraction

Poster Images: Explicitly Specified Preview Images

Ideally, the plugin explicitly specifies a preview image to use. This can be done by adding an optional **poster** attribute to the plugin embedding object tag. Here's an example:

```
<object data="http://example.com/flash.swf"
        type="application/x-shockwave-flash"
        poster="poster.png">
</object>
```

The attribute is named **poster**, as that's the HTML5 attribute name used in video tags for the same purpose. This attribute is allowed, as the [W3C spec for the object tag](#) allows for extra arbitrary attributes to be passed directly to the plugin.

The value of the poster param will be interpreted the same way as the *srcset* attribute of an *img* tag. This is to support faithful reproduction on high-dpi displays. Here's an example of the *srcset* syntax:

```
<object data="http://example.com/flash.swf"
        type="application/x-shockwave-flash"
        poster="snapshot1x.png 1x, snapshot2x.png 2x">
</object>
```

When there is an explicitly specified poster image, we should not load the actual plugin until the user clicks on the poster.



When there is a size mismatch for the poster image, it is scaled in the same way as the video tag poster, following the CSS *contain* rules. The image is scaled to the same size as the plugin container, preserving the aspect ratio and containing the whole height and width within.

Preroll Keyframe Extraction



If there is no explicitly specified preview image, we have to use heuristics to guess a representative keyframe to use as a preview image.

Extracting a good preview image is not always easy. Here are some “bad” frames to choose:

1. Blank, uninteresting frames. A plugin will often render a few blank frames while it is loading internal resources or waiting for the page to finish loading. We don't want to use those, as they don't tell the user what to expect.
2. Animated transitions. Peripheral content often animate from black to the actual image via a blur or swipe animation. These frames look 'interesting' from an entropy calculation, but are not useful for humans, as they are often unreadable.

Bad, unreadable:	
Good:	

3. “Bad” loading frames. While keeping “good” loading frames. This might be impossibly hard, unfortunately.

<p>Bad</p> 	<p>Good</p> 
--	--

Chrome currently simply runs the plugin and looks for the very first frame that’s deemed “interesting” via a simple luma histogram calculation. Any frame that’s not a solid color usually passes this test. We chose a simplistic implementation for performance reasons.

Plugin Pausing and Resuming

When there is a poster image available, (when there is an explicitly specified one), this is a non-issue. We will simply defer launching the plugin until the user clicks the preview image.

In the frequent case when we do not have a preview image already available, we will have to run the plugin for a short period of time to view rendered frames and extract a suitable preview.

After that, we need to either throttle the plugin until the user clicks on it, or kill it entirely. Chrome currently implements the throttling technique described below, to avoid interfering with JS/plugin communication.

Throttling using Fake Offscreen Technique

To throttle a plugin, Chrome tells the plugin it is not visible (background tab / below the fold). This is done through Pepper’s PPB_View API. **This works great most of the time.** Plugins stop rendering and CPU/GPU usage drops dramatically.

According to the [Adobe reference](#), recent versions of Flash will throttle when in a background

tab or out of viewport. The reference states that plugins will continue to run at an internal 4fps frame rate. The rendering, however, will be halted, so the user won't actually see 4fps being rendered to the screen.

For animated SWFs, the animation appears to be truly suspended. When the user clicks the plugin to resume, the animation picks up right where it left off. This case covers most peripheral Flash content that we reviewed.

Throttling audio and network access

Plugins with audio continue playing audio when they receive an offscreen event. They also continue to access the network to keep any streaming buffers filled. This is desirable and expected for when such plugins are actually offscreen.

However, this is problematic for throttled plugins that have received a "fake" offscreen event. The video rendering stops and the plugin is greyed out, but the audio continues. Additionally, when the user clicks on the throttled plugin, the video 'snaps' to the new time instantly. This behavior would be surprising and annoying.

Therefore, we implemented an approach that halts audio and network traffic:

1. Defer the start of the periodic audio callback defined in PPB_Audio spec. This will not only mute the audio output, but will also cause audio buffer pressure. This causes some videos to also pause video playback until the audio buffer pressure is cleared.
2. Block TCPSocket::Read commands. This will prevent transmission of streaming content. This should also block streaming content playback. This is also necessary to prevent RTMP video from continuing to stream the video and fill up the user's memory.

Benefits of the Fake Offscreen Technique:

- No scripting issues.
- SWFs should get a ThrottleEvent from Flash, and can reactively use this signal to park their resource usage further.
- No need to relaunch plugins.

Potential issues:

- Doesn't save as much power as outright killing it. Audio / video plugins keep running and streaming content from the network.
- Chrome doesn't control the throttling mechanism. We rely on the plugin author's native throttling.
- There can be other unforeseen side effects of providing fake view clip information to the plugin. This also is a bit exploitative of the Pepper API.

Rollout

The plugin setting is under Preferences->Content Settings->Plugins.

For users that have never explicitly chosen a setting within the Preferences menu, we control the default behavior using the Field Trial servers. If the rollout goes badly, we can use Field Trial servers to remotely restore all these users back to the old Play All behavior.

If the user has local command line flags, an enterprise policy, or has explicitly made a choice within the Preferences menu, we will respect that and our Field Trial servers will have no effect.

The default behavior will never be stored in prefs file. This could possibly be misinterpreted as a user choice, and hamper our ability to reset the default behavior in the case of rollback.

Project Information

Contact:

groby@chromium.org (TLM)

laforge@chromium.org (TPM)

tommycli@chromium.org (SWE)

Security Considerations

We are removing *Click to Play* option for plugins in Chrome settings. This is a security win, as *Click to Play* provides the illusion of security, but is trivially clickjackable. We migrated existing *Click to Play* users to *Block*.

Power Benchmarks

Preliminary benchmark results are accessible in [this document](#) (internal-only).