Multiple Highlights and Dual Screen Hinge Overlay

Attention: Externally visible, non-confidential

Author: brgoddar@microsoft.com; soxia@microsoft.com; shanejc@microsoft.com

Status: Draft

Created: 2020-02-06 / **Last Updated**: 2020-02-12

One-page overview

Summary

This explainer proposes CDP methods change for dual screen support. It also discusses plans to allow showing and hiding of multiple instances of a highlight, as well as the backend support to allow retaining a highlight in different inspect mode.

Platforms

All

Team

Songtao (Stan) Xia, soxia@microsoft.com

Brandon Goddard, brgoddar@microsoft.com

Shane Clifford, shanejc@microsoft.com

Tracking issue

CrBug Link

Value proposition

To enable the correct drawing of a hinge of a dual screen device, we propose changes to the Overlay domain as well as refactoring at the backend. We also discuss the future plan to support multiple highlights: how the api should be structured to support adding or removing highlights and how to stack them in order.

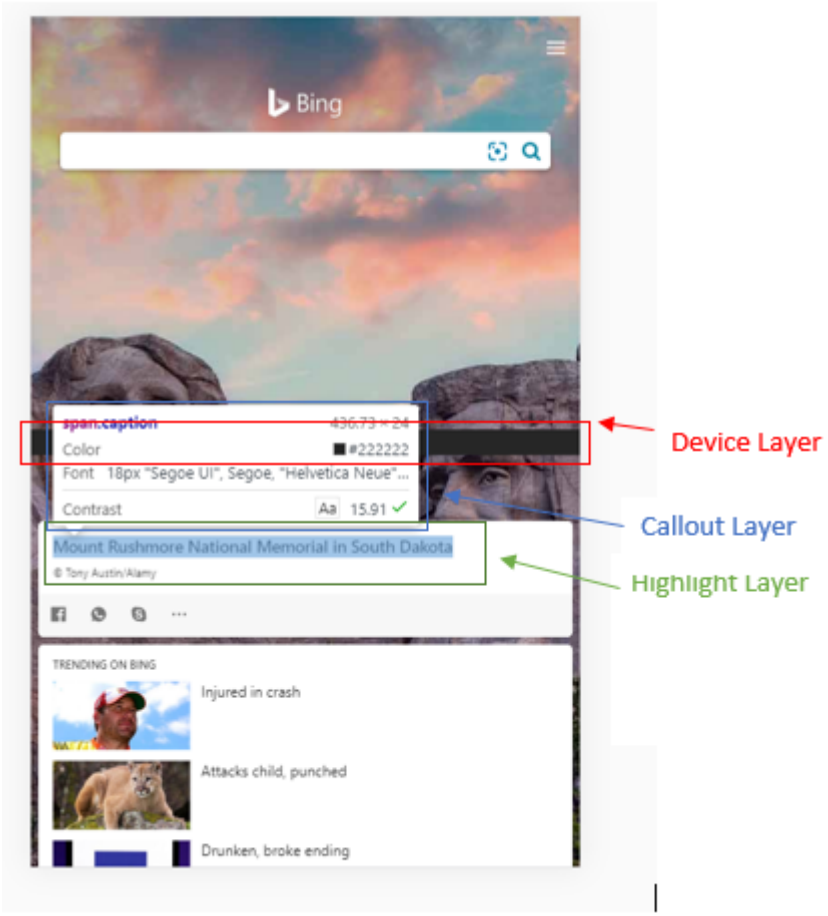

Figure 1: Multiple Overlays

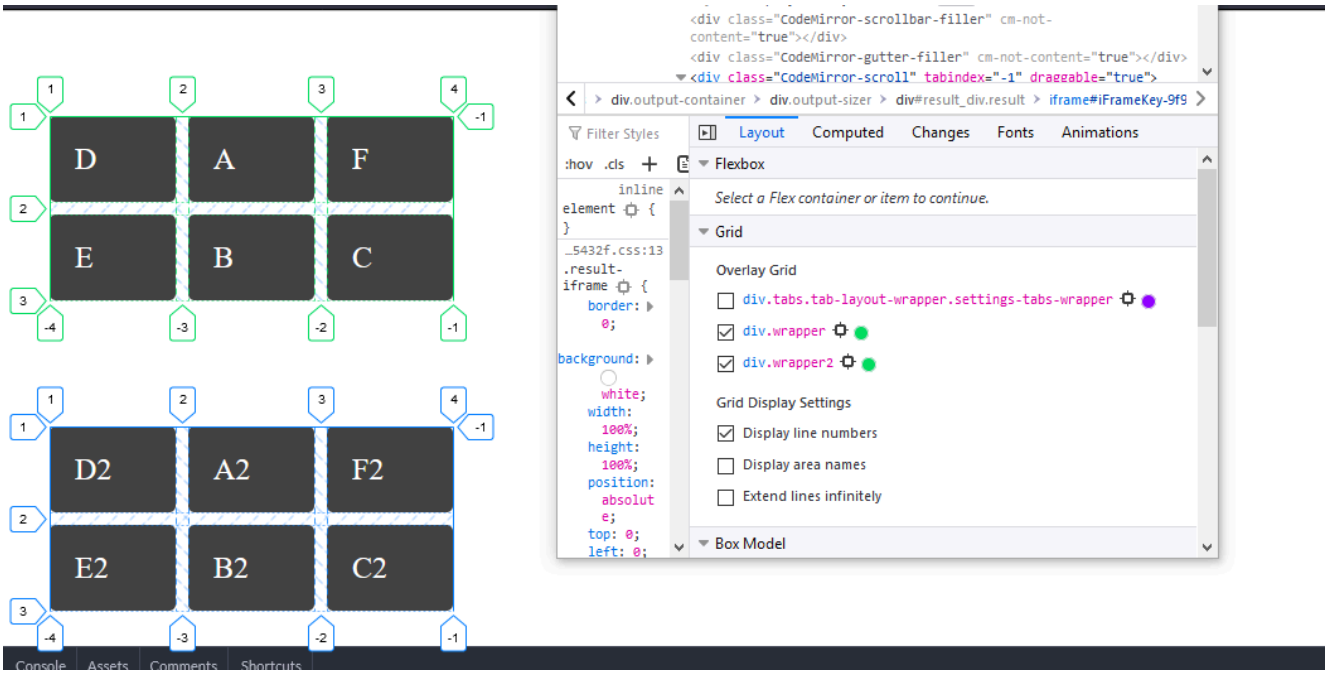

Figure 2: Example Grid Tooling from Firefox: multiple grids highlighted independently

Code affected

- DevTools frontend
- Backend Overlay domain
- Overlay Domain CDP (a new method).

Chromium WIP PR

A PR to illustrate the ideas of this explainer is located at:

https://chromium-review.googlesource.com/c/chromium/src/+/2087421

Signed off by

Name	Write (not) LGTM in this row
aerotwist@chromium.org	
bmeurer@chromium.org	LGTM
leolee@microsoft.com	LGTM
Rob.Paveza@microsoft.com	
yangguo@chromium.org	lgtm
mathias@chromium.org	LGTM with caseq's concerns addressed
changhaohan@chromium.org	LGTM
dgozman@microsoft.com	LGTM
caseq@chromium.org	setShowHinge lgtm

Design

Background

One example of multiple overlays in DevTools is when we have the node highlighting in dual screen emulation. We will use this as the running example throughout the explainer. A user may elect to

enter or exit dual screen mode, in which a hinge that separates the two screens may be rendered. Independently, the user may switch the inspector mode on and off; in the inspector mode, parts of the page may be highlighted, as illustrated in Figure.1. We expect, among other things:

- Exiting the inspector mode when in dual screen mode leaves the hinge in place; exiting the dual screen mode leaves the node highlight, if any, in place.
- The node outline (in green rectangle) hides behind the hinge (in red rectangle) if they overlap; the element info (in blue rectangle) covers part of the hinge if they overlap.

We also recognize that sometimes we need to highlight multiple instances of the same kind of element. For example, in Figure 2, we highlight multiple CSS grids, but can highlight each individual grid separately. Support of grid tooling highlights (i.e., adding new CDP methods) is the subject of a different explainer. The discussion of how this is supported, which is included in discussion sections in the document, helped to shape the design here.

Different drawings on top of the web page are managed by the Overlay domain. To draw an overlay, the frontend sends CDP messages to the backend. There are some problems to be addressed:

- There are no CDP methods directly suitable for drawing or hiding a hinge.
- a hinge or a grid tooling highlight needs to be present across inspect mode change, and properly drawn when other overlays.
- The hiding of a hinge or a grid tooling, or the hiding of other overlays with a hinge present, needs to be properly handled.
- Some highlights, notably grid tooling, may have multiple instances.
- We need a design to ensure the stacking order of overlays.

The details of supporting grid tooling highlights is the topic of a separate explainer. This explainer focuses on dual screen hinge support while discussing the issues above. The proposed solution here only requires minimal change to the CDP domain and some refactoring of the backend. We first present the changes to CDP methods.

CDP Changes

There is only a small change to the CDP Overlay domain: one method to support a hinge is added and the semantics of the setInspectMode method is made clearer in our context.

Below we will describe a few (backend) concepts related to how we manage multiple overlays and to how we plan to manage the stacking order of a set of overlays. Then we list the changes to the CDP for hinge support. Finally, we provide an example of using our proposed change to work on our example.

Managing Overlays

We use the concept of tool stack to manage overlays. A tool stack manages the lifetime, rendering, and event handling of multiple *tools.* Each tool draws one kind of overlay. A tool can be added to or removed from a tool stack. At a given time, what a user sees on the screen is the collective drawing of all tool instances in a tool stack.

In our example, when the frontend is in the inspector view, a new tool stack starts, possibly not known to the frontend. Initially no tool is in place. If the frontend enters the dual screen mode, the tool stack

will contain one tool, responsible for rendering a hinge. If then the frontend selects inspect mode, then possibly a node highlight tool is added to the tool stack. If after that, we leave the dual screen mode, the highlight tool is still in the tool stack but the hinge tool is not, and vice versa.

Changes to the Overlay domain related to the tools include:

- a new method for adding and removing a dual screen hinge tool.

Details are presented later in the section.

Discussion on Multiple Tool Instances

Most tools have only one instance. But certain highlight tools, such as grid tooling highlights, may have multiple instances. For such tools, the plan is for the tool to manage its own tool instances. The CDP API should allow a tool specific method to add or remove an instance of this kind. For example, we may have a `highlightGrid(selector)` where a selector decides which grid to highlight. This gives us a venue to experiment with highlighting different objects. A general removal method is another option with a simpler API and more flexibility. Using tool-specific adding and removal methods, we can gather more knowledge about the usage of such APIs and decide if we want the general version in the future.

Stacking Order

From the backend point of view, all the overlays in a tool stack are rendered on a single html. We control the stacking order by hardcoding the z-index at the backend. There is a design that allows some reuse of a tool with a different hardcoded z-index. For example, the highlight rect tool, draw a rectangle normally at z-index 1. If we use highlight rect to draw a hinge, which is above the normal rectangles, we may create a subclass, called a hinge tool for example, and hardcode its z-index at 5.

Discussion on Z-Indices

It is understood that the approach above is not going to scale but should satisfy our needs in the near future. If later we find the approach too restrictive, or the number of (tool, z-index) pairs becomes too large to manage, we may give the frontend more control over the backend, including moving the UI logic to the frontend all together.

Semantic Change to Existing CDP Methods

`SetInspectMode`'s semantics will change but existing calls still work. It now removes all the tools not configured to exist in the new mode (including `None`), creates a tool corresponding to the mode if needed,, and adds the newly created to the tool stack.

To support mode change, tools are required to indicate whether they should be kept when switching to a new mode.

New Method: Hinge Support

We propose a method below to allow the frontend to explicitly control the lifetime of a hinge tool and a method for removing named highlights.

Method	Description
`setShowHinge(optional HingeConfig hingeConfig)`	Create a hinge tool and add it to the current tool stack. `hingeConfig` may further contain information needed to construct the tool, like position, color, gradient, etc. When not provided, the hinge is hidden.

For example, if we want to create a hinge tool, the call would be like:

```
overlayAgent.setShowHinge({x:hingeLocX, y: hingeLocY, width: hingeWidth,
height: hingeHeight});
```

One may have noticed that we do not have methods to start or end a tool stack. We plan to use a default tool stack associated with a view. For example, inspect view can be in an inspect view tool stack. At this point, we do not see the need to add a tool stack explicitly.

CDPs in Example

In our running example, a possible sequence of CDP calls are:

1. If we enter dual screen mode, we will call `setShowHinge` to add the hinge.

```
overlayAgent.setShowHinge({x: hx, y: hy, width: hw, height: hh});
```

2. If we select a node, we call `highlightNode` to highlight a node, or maybe for some Android dual screen devices, we highlight a node covered by the hinge,

```
overlayAgent.highlightNode(highlightConfig, nodeID);
```

3a. We call `setShowHinge` with no parameter to stop drawing the hinge if we exit the dual screen mode.

```
overlayAgent.setShowHinge();
```

3b. We call `setInspectMode(None)` if we exit the inspector mode. Highlighting node tool reports that it should not be in inspect mode `None`, so it goes away. The hinge is consulted and it will remain.

3c. We call hideHighlight if we do not have a node to draw but still in the inspector mode, note this call removes all highlights.

```
overlayAgent.hideHighlight();
```

Like discussed above, we may extend in the future highlightNode to select which node highlights we want to show or remove. This is not covered in this explainer.

Now we discuss the changes to the backend, which is necessary for the implementation of the extension we made with CDP methods.

Change to Backend

The changes to the backend is: 1) to split the `InspectTool` class into an `ToolStack` class and a new, role-reduced, `InspectTool` class, and 2) for the InspectOverlayAgent to use a tool stack to render overlays or to relay input processing if necessary.

The `InspectTool` class in the previous implementation is a class which:

- is associated with a placeholder html,
- may elect to handle input events, and
- is responsible for the rendering of all the current overlay.

We propose to reduce the responsibility of an `InspectTool` to be rendering one overlay in a tool stack.

A `ToolStack` class manages the overlays drawn by a frontend view. The `ToolStack`'s responsibility is:

- is associated with a placeholder html,
- manages a collection of `InspectTools`,
- may query the `InspectTools` to see if any of them handles input events, and
- is responsible for all the rendering of the current set of overlays, in their z-index order.

An `InspectTool` now:

- can be added to or removed from a `ToolStack`,
- decide whether it should be kept in an inspect mode,
- renders an overlay, may allow configuration of its z-index, and,
- may elect to handle input events when queried by the tool stack.

We may want to refactor some of the current tools to allow further reuse. But that is not the priority of this explainer.

Core user stories

As described by the running example, when we have both dual screen mode and different inspect mode, we expect the changing of one mode does not interfere with the rendering of overlays from the other mode.

A user may be able to select and highlight multiple grids in grid tooling.

Rollout plan

We will implement this feature behind a feature flag. Once it receives an adequate level of testing and ensures we aren't regressing any of the Core principle considerations, we will turn it on by default.

This will enable us to ship features that build on top of this feature, like dual-screen emulation, without having customers to set two flags.

Core principle considerations

Speed

The impact of the revision on performance is predicted to be insignificant. There can be extra CDP calls from starting or removing an overlay and especially from the new possibility of structuring an overlay scenario as multiple tools. But we don't expect the number of overlays to be high and the situation of multiple overlays to be frequently seen. There could also be more calls to `evaluateInOverlay` in the backend. But again the impact is not expected to be huge.

Security

This feature will not impact the security of the Chromium browser or the DevTools.

Simplicity

This feature enables developers who build the DevTools to compose multiple overlays in an intuitive and predictable way. The hacks required for composing overlays today are prone to bugs, so this feature will meet developers' expectations for consistent and deterministic overlays.

Accessibility

We will ensure that we do not regress the current accessibility of overlays and hope to improve the accessibility of overlays with this feature as well.

Testing plan

We shall implement some web_tests. Other forms of integration tests are being considered.

Followup work

Certain tools in the Overlay backend may be refactored to encourage code reuse and to allow a tool to be better suited for Stackable Overlay. Also, in the Javascript code in the Overlay backend, some functionality can be refactored.
