placeElement

Fernando Serboncini <fserb@google.com>
Khushal Sagar <khushalsagar@google.com>

This proposal covers APIs to allow live HTML elements on Canvas 2D, WebGL and WebGPU.

Use Cases

- **Styled, Layouted & Accessible Text in Canvas.** There's a strong need for better text support on Canvas. This includes both visual features like multi-line styled text but also the possibility to support the same level of user interaction as the rest of the web (scrolling, interactions, accessibility, indexability, translate, find-in-page, IME input, spellcheck, autofill, etc).
- **Interactive forms.** Access to live interactive forms, links, editable content with the same quality as the web. Close the app gap with Flash.
- **Composing HTML elements with shaders.** "CSS shaders" precede WebGL. The ability to use shader effects with real HTML.
- **Allow HTML rendering in 3D Context.** To be able to have interfaces and full accessible content in 3D.

We also got strong public feedback on some demos for this feature.

Explainer

There are 2 API surfaces to be exposed on this proposal. First, a high level API that brings an Element and its subtree into the 2D Canvas. Second, a broken

down version that allows finer control over Javascript and is also available in 3D contexts.

For this explainer, we use the term “live element” to describe an HTML element not only rendered to the screen, but that also responds to events, user interactions, text selection, scrolling, tab order, accessibility.

placeElement

placeElement is the high level “do the right thing” API.

```JavaScript
interface mixin CanvasPlaceElements {
  Element placeElement(Element el, double x, double y);
}

CanvasRenderingContext2D includes CanvasPlaceElements;
```

This element must be a (direct?) child of the Canvas element. The children elements of Canvas don’t impact the overall document layout and, before placeElement, are considered fallback content and ignored on modern browsers. Once placeElement is called, the element becomes alive until the canvas is reset (either by ctx.reset() or canvas.width = canvas.width) or until it’s not a child of the canvas anymore.

The element is rendered at a particular position and takes the CTM (current transform matrix) of the canvas into consideration.

The element gets repainted (together with the canvas) as needed. The call order of placeElement implicitly describes which canvas draw commands happen before (below) or after (above) the rendering of the element.

placeElement() may taint the canvas. It returns the element placed or null if failed.

It’s also worth noting that this never duplicates the element. If called twice on a single canvas, it simply “replaces” (repositions) the element to a new location + CTM and to a new position in the canvas stack.

Usage example:

```
JavaScript
<!doctype html>
<html><body>
<canvas id=c>
  <div id=d>hello <a href="https://example.com">world</a>!</div>
</canvas>
<script>
const ctx = document.getElementById("c").getContext("2d");
ctx.rotate(Math.PI / 4);
ctx.placeElement(document.getElementById("d"), 10, 10);
</script>
</body></html>
```

This would add a text “hello world!” to the canvas, with a clickable link and interactable text rotated by 45 degrees.

drawElement

The second API is a broken down version of the previous one, that allows the same behavior as placeElement, but broken down in stages. It requires more work from developers to support live elements, but is also exposed to 3D contexts, which placeElement isn’t. This API is also useful for cases where interaction is not required, like drawing better text for images or for screenshotting the page.

```
JavaScript

interface mixin CanvasDrawElements {
  undefined updateElement(Element el,
    optional DOMMatrixInit transform = {}, optional long zOrder);
```

```
  undefined removeElement(Element el);
}

interface CanvasInvalidationEvent {
  readonly attribute HTMLCanvasElement canvas;
  readonly attribute DOMString reason;
  readonly attribute DOMRect invalidation;
}

// for Canvas 2D
// drawImage
typedef (... or Element) CanvasImageSource;

CanvasRenderingContext2D includes CanvasDrawElements;

// for WebGL
// texImage2D, texSubImage2D
typedef (... or Element) TexImageSource;

WebGLRenderingContext includes CanvasDrawElements;
WebGL2RenderingContext includes CanvasDrawElements;

// for WebGPU
// copyExternalImageToTexture
typedef (... or Element) GPUImageCopyExternalImageSource;

GPUCanvasContext includes CanvasDrawElements;
```

When using the drawElement API, the user has to complete the loop to make the element alive in Javascript:

- it needs to call the draw function (drawImage, texImage2D, GPUImageCopyExternalImageSource),

- update the element transform (so the browser knows where the element ended up (in relationship to the canvas), and
- respond to a new invalidation event (for redrawing or refocusing within the scene, as needed). In theory, the invalidation event is optional if the user is updating the element on RAF, but it could still be useful if the page wants to respond to a find-in-page event, for example.

In theory, drawElement can be used to provide non-accessible text. We still enforce that the element (at drawing time) must be a child of its canvas, but the liveness of the element depends on developers doing the right thing. That said, the current status quo is that it's impossible for developers to "do the right thing", i.e., text in 3D contexts - for example - is currently always inaccessible. This API would allow developers to do the right thing.

An already placedElement can be drawn, but cannot have its transform updated.

Usage example:

```
JavaScript
<!doctype html>
<html><body>
<canvas id=c>
  <div id=d>hello <a href="https://example.com">world</a>!</div>
</canvas>
<script>
const ctx = document.getElementById("c").getContext("2d");
const el = document.getElementById("d");
ctx.rotate(Math.PI / 4);
ctx.drawImage(el, 10, 10);
ctx.updatedElement(el, ctx.getTransform());
</script>
</body></html>
```

This would render the text "hello world!" to the canvas with an interactable text.

Potential Challenges

Privacy-preserving rendering

placeElement can taint the canvas2D as needed, so even cross-origin content could eventually be placed without privacy issues. For drawElement, there's no tainting mechanism for WebGL/WebGPU but even if it existed, the existence of shaders makes it that all rendered content should be assumed to be readable by the page. This is why WebGL/WebGPU today disallows rendering cross-origin images.

To support drawElement at all, we need to make sure that the rendering of the elements is done in a privacy-preserving way. Visited links may be changing soon to be domain-based. But other issues (like dictionary spelling) may need to be disabled.

Complexity of rendering certain elements

It's possible that some elements may not be renderable to a texture (2D or 3D). For those cases, we are ready to have an initial disallowed list of elements to simplify implementation by UAs. For those cases, placeElement() could return null to indicate this placeElement is not supported.

Re-compositing for placeElement

placeElement requires the browser to recompose the canvas buffer each time the element changes. This should be equivalent to rendering all commands executed on the canvas with the latest contents for each placed element. If implemented as such, this would be inefficient since the buffer of commands can grow unbounded.

The browser could optimize by squashing sets of commands into buffers at placeElement boundaries whenever possible (each saveLayer would also need to split up?).

Hit Testing Data structures

The canvas subtree is laid out based on the DOM/style applied to the elements. However, their contents are not painted based on the position or ordering of their boxes. It is instead defined by the canvas commands which render the element's image into

the canvas buffer.

This affects both hit-testing and behaviour of platform APIs (like IntersectionObserver) which allows authors to observe the position of these elements. This is the data provided by CanvasDrawElements::updateElement API for drawElement mode and done automatically by the browser for placeElement mode.

Implementation Details

Here is a summary of the implementation details of this spec in Chromium. Its function is to illustrate some of the potential challenges of a placeElement implementation.

Render Canvas Descendants

The following changes are needed to layout and paint canvas descendants so their image can be used by drawImage/texImage2D/externalTexImage2D:

- Layout for canvas children
  Generally children of replaced elements are ignored by layout. The exception to this has been video elements which use a shadow DOM to render controls: LayoutMedia::CanHaveChildren. If a canvas uses the proposed APIs above, this mode will be supported for canvas as well.

- Privacy preserving rendering.
  Visited links, dictionary spell checking, etc.

- Painting
  The canvas subtree goes through the pre-paint step during the regular lifecycle update as usual (except for update element position below). However the subtree is skipped during paint when preparing the display list forwarded to the compositor.

  Instead an independent paint step is triggered similar to DataTransfer::NodeImage to generate a PaintRecord for the element which is appended to the canvas display list (for 2D) or rasterized to a texture (for 3D).

This step will need to support caching the painted output.

Make Elements Interactive/Animated

This is the set of changes required to make elements rendered onto the canvas interactive and animated.

- Invalidation signal
  placedElements needs to signal to canvas that it needs to redraw itself; drawElements needs to trigger an event to tell JS that the element has been updated.

- Update element position
  From a position perspective, placeElement() is equivalent to drawElement with updateElement() being called with the canvas' CTM at the time of placeElement(). Both those paths need to update the element's "final real position". In Chrome, this was done through a transform paint property node, that is equivalent to what other CSS properties (like transform) generate.

- Update element stacking
  Similar to above, generally hit-testing depends on the order for paint layers to decide which element draws on top. This stacking is generated based on the DOM order + styles (z-order) applied to the elements. In Chrome, this is done by modifying the paint layer iteration order based on the stacking for placeElement(), or the zOrder provided by updateElement().

- Forcing main thread animations
  For drawElement interactive mode (used by 3D) all animations and interaction needs to be forced to run on the main thread. This includes the following features:
    - Composited animations like transform, opacity, filter etc.
    - Scrolling
    - Off thread paint worklets
    - Video

  There are already code-paths to run these on the main thread in some cases which can be reused. But some features are only supported in the compositor:

animation images, OOPIFs. Animated image can be main thread driven, so support can be added incrementally. OOPIF support is not needed due to privacy restrictions.

- Hit Testing in the compositor
  The compositor gets hit test regions to allow faster interaction without going back to the main thread. For example, regions which have a non-passive touch handler or main thread scrolling regions. This data will need to be added to the canvas's CC layer for placed elements.

Make Canvas2D auto-redraw on placeElement

For Canvas2D to support "auto redrawing" when its placed element has updated, there were a couple techniques we considered to combine the rendering of placed elements with regular canvas drawing.

Currently each canvas is represented as a single TextureLayer in the compositing stack. The paint commands added to the canvas are serialized directly from the renderer main thread to the GPU process to update the texture referenced by this layer, which composites it with the rest of the Document.

1. Display list with references. Keep canvas drawing commands as a display list and add references to where placeElement rendering should go. At rendering time, build a final display list and render it.

   This is the simplest approach but can be inefficient because canvas needs to persist all commands added to it since the last reset.

2. Layered canvas. When there's a placeElement call, all previous canvas2D drawings get collapsed into a texture. Canvas manages this list of textures + placeElement rendering and composites them in order.

   This is an optimization to the approach above. It allows for re-presenting the canvas with the updated elements without redrawing all the canvas commands.

3. Layered compositing. Similar to layered canvas, but instead of canvas managing the final compositing, it is deferred to the compositor. The canvas is represented as a list of layers to the compositor instead of a single layer.

This list is made up of TextureLayers (for collapsed commands) and the placedElement's contents are represented by a list of layers produced by the paint system similar to the regular rendering of web content. The main pro of this approach is threaded rendering/scrolling since placed elements are rendered similar to non-canvas elements.

Note: Both #1 and #2 will require main thread invalidation + redraw of the canvas using the same signal provided to script. #3 can use the regular paint invalidation code-path to commit a new display list to the compositor.

Open Issues

1. Should placed elements be direct children of the canvas or any descendant is ok? Might be harder to fake the element's rendered position in the canvas if there are boxes between the canvas and the element in the tree.
2. How are non-placed canvas children represented in the layout tree. We likely need to layout the entire subtree (especially if a placed element can be a descendant) and make everything not placed visibility: hidden so they are not painted and inert.
3. Should we force containment (layout or paint) on canvas to ensure the children don't affect the layout of the rest of the Document?
4. Is it worth providing the invalidation rect (based on element bounds) or will authors always do full redraw?
5. Should draw commands after placeElement block the interaction rect of elements (since they would occlude the element)? Should there be a context state to control this behavior?
6. When painting the elements, which box should be painted? Should it include shadow? If so, how does the developer know what is the full rendered size?
7. If developers want to align placeElements (either by baseline or box), how are they supposed to do it?
8. What happens to placed inline elements?
9. Clarify the timing for when the image is generated when using drawImage. Ideally it should be at the end of all script callbacks during update the rendering but unclear whether that's possible for 3D.
10. The author might draw a subset of the element. Should we add a "src rect" to updateElement.

11. Should we allow a place element to be a descendant of another place element? It will cause double draw of the descendant and also make it harder to reason about the element's onscreen position.
12. Placed elements must be stacking contexts. Do we need any other restriction?
13. Should we force blockifying canvas children?