CSS View Transitions Module Level 2

W3C Working Draft, 13 November 2024

▼ More details about this document

This version:
https://www.w3.org/TR/2024/WD-css-view-transitions-2-20241113/

Latest published version:
https://www.w3.org/TR/css-view-transitions-2/

Editor's Draft:
https://drafts.csswg.org/css-view-transitions-2/

History:
https://www.w3.org/standards/history/css-view-transitions-2

Feedback:
CSSWG Issues Repository
Inline In Spec

Editors:
Noam Rosenthal (Google)
Khushal Sagar (Google)
Vladimir Levin (Google)
Tab Atkins-Bittner (Google)

Suggest an Edit for this Spec:
GitHub Editor

Abstract

This module defines how the View Transition API works with cross-document navigations.

CSS is a language for describing the rendering of structured documents (such as HTML and XML) on screen, on paper, etc.

Status of this document

This section describes the status of this document at the time of its publication. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

This document was published by the CSS Working Group as a **Working Draft** using the Recommendation track.
Publication as a Working Draft does not imply endorsement by W3C and its Members.

This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

Please send feedback by filing issues in GitHub (preferred), including the spec code "css-view-transitions" in the title, like this: "[css-view-transitions] *…summary of comment…*". All issues and comments are archived. Alternately, feedback can be sent to the (archived) public mailing list www-style@w3.org.

This document is governed by the 03 November 2023 W3C Process Document.

This document was produced by a group operating under the W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Table of Contents

1. Introduction

This section is non-normative.

View Transitions, as specified in [css-view-transitions-1], is a feature that allows developers to create animated transitions between visual states of the document.

Level 2 extends that specification, by adding the necessary API and lifecycle to enable transitions across a same-origin cross-document navigation, as well as a few additions that make it easier to author pages with richer view transitions.

Level 2 defines the following:

- Cross-document view transitions, including the '@view-transition' rule and the algorithms that enable the cross-document view transition lifecycle.
- Selective view transitions, a way to match styles based on the existence of an active view transition, and more specifically based on the active view transition being of a certain type.
- Sharing styles between view transition pseudo-elements, a way to declare a style once, and use it for multiple view transition pseudo-elements. This includes the 'view-transition-class' property, and additions to named pseudo-elements

2. Cross-document view transitions

2.1. Overview

This section is non-normative.

2.1.1. Activation

With same-document view transitions, the author starts a view transition using JavaScript, by calling `startViewTransition`. In cross-document view transition, what triggers a view transition is a navigation between two documents, as long as the following conditions are met:

- Both documents are of the same origin;
- The page is visible throughout the entire course of the navigation;
- The user initiates the navigation by interacting with the page, e.g. by clicking a link or submitting a form; or by interacting with the browser UI to do a `traverse` navigation (back/forward). This excludes, for example, navigations initiated by the URL bar;
- the navigation didn't include cross-origin redirects; and
- both documents opted in to cross-document view transitions, using the '@view-transition' rule.

See the lifecycle section for more details.

2.1.2. Waiting for the new state to stabilize

In same-document view transitions, the author can indicate when the new state of the transition is in a stable state by using the callback passed to `startViewTransition`. Since cross-document view transitions are declarative, there is no such explicit promise. Instead, the user agent relies on the render-blocking mechanism to decide when the document has reached a stable state. In this way, the author can use the `blocking` attribute, to delay the transition until:

- All expected scripts are executed, by using the script's `blocking` attribute on required scripts.
- All expected styles are executed, by using the style or link's `blocking` attribute on required styles.
- All expected HTML elements are seen by the parser, using an `expect` `HTMLLinkElement` element.

> NOTE: overusing the render-blocking mechanism could make it so that the old state remains frozen for a long time, resulting in a jarring user experience. To avoid this, it's advised to ensure that the render-blocking elements are available in a timely manner.

EXAMPLE 1

In this example, the last frame of the old document will be shown, and the animation will be delayed, until all the following conditions are met:

- `style.css` is applied, to ensure the new state has the correct styles
- `fixup.js` is run, to ensure the presentation is up to date with script-based fixups.
- The `main-article` section is seen and parsed, to ensure enough of the content is loaded before allowing the transition to proceed.

```
<!DOCTYPE html>
<html>
  <head>
    < !-- This will be render-blocking by default -->
    <link rel="stylesheet" href="style.css">

    < !-- Since this script fixes up the layout, marking it as render blocking will
        ensure it's run before the view transition is activated -->
    <script async href="fixup.js" blocking="render"></script>

    < !-- Wait until the main-article element is seen and fully parsed before
        activating the transition -->
    <link rel="expect" href="#main-article" blocking="render">
  </head>
  <body>
    <header>...</header>
    <main>
      <article id="main-article">...</article>
    </main>
    <article id="secondary-article">...</article>
  </body>
</html>
```

2.1.3. Customization

The `ViewTransition` object enables customizing the transition in script. Same-document view transitions use a single `ViewTransition` object returned from the `startViewTransition` call for the entire lifecycle. Cross-document view transitions have two `ViewTransition` objects, one in the old document and one in the new document.

2.1.3.1. Handling the view transition in the old document

The `pageswap` event is fired at the last moment before a document is about to be unloaded and swapped by another document. It can be used to find out whether a view transition is about to take place, customize it using `types`, make last minute changes to the captured elements, or skip it if necessary. The `PageSwapEvent` interface has a `viewTransition` object, which would be non-null when the navigation is eligible to a view transition, and a `activation` object, providing handy information about the navigation, like the URL after redirects. The transition's `finished` promise can be used for cleaning up after the transition, in case the document is later restored from BFCache.

2.1.3.2. Handling the view transition in the new document

The `pagereveal` event is fired right before presenting the first frame of the new document. It can be used to find out if the view transition is still valid, by querying the `viewTransition` attribute. Similar to a same-document view transition, the author can now select different `types`, make last minute changes to the captured elements, wait for the transition to be `ready` in order to animate it, or skip it altogether.

2.1.4. Lifecycle

This section is non-normative.

A successful cross-document view transition goes through the following phases:

1. In the old `Document`:
    1. The user initiates a navigation, by clicking a link, submitting a form, pressing the browser's back button, etc.

        NOTE: some navigations do not trigger a view-transition, e.g. typing a new address in the URL bar.

    2. When the new `Document` is ready to be activated, the `pageswap` event is fired.
    3. If the navigation is same origin, has no cross-origin redirects, and the old `Document` has opted in to cross-document view transitions, the event's `viewTransition` attribute would be a `ViewTransition` object.
    4. The author can now customize the transition, e.g. by mutating its `types`, or `skip` it altogether.
    5. If the `ViewTransition` is not skipped, the state of the old document is captured.
    6. The navigation proceeds: the old `Document` is unloaded, and the new `Document` is now active.
2. Then, in the new `Document`:
    1. When the new `Document` is ready for its first rendering opportunity, an event named `pagereveal` is fired on the new `Document`, with a `viewTransition` attribute.
    2. This `ViewTransition`'s `updateCallbackDone` promise is already resolved, and its captured elements are populated from the old `Document`.

3. This is another opportunity for the author to customize the transition, e.g. by mutating its `types`, or `skip` it altogether.
4. The state of the new document is captured as the "new" state of the transition.
5. From this point forward, the transition continues in a similar fashion to a same-document transition, as per activate view transition.

§ 2.2. Examples

EXAMPLE 2 ¶

To generate the same cross-fade as in the first example *CSS View Transitions 1* § 1.6 Examples, but across documents, we don't need JavaScript.

Instead, we opt in to triggering view-transitions on navigation in both page 1 and page 2:

```
// in both documents:
@view-transition {
  navigation: auto;
}
```

A link from page 1 to or from page 2 would generate a crossfade transition for example 1. To achieve the effect examples 2, 3 & 4, simply put the CSS for the pseudo-elements in both documents.

EXAMPLE 3 ¶

Note that the '@view-transition' rule can be used together with media queries. For example, this would only perform the transition when the screen width is greater than:

```
@view-transition {
  navigation: auto;
}

@media (max-width: 600px) {
  navigation: none;
}
```

EXAMPLE 4 ¶

To achieve the effect in example 5, we have to do several things:

- Opt-in to navigation-triggered view-transitions in both pages.
- Pass the click location to the new document, e.g. via `sessionStorage`.
- Intercept the `ViewTransition` object in the new document, using the `pagereveal` event.

In both pages:

```
@view-transition {
  navigation: auto;
}
```

In the old page:

```
addEventListener('click', event => {
  sessionStorage.setItem("lastClickX", event.clientX);
  sessionStorage.setItem("lastClickY", event.clientY);
});
```

In the new page:

```
// This would run both on initial load and on reactivation from BFCache.
addEventListener("pagereveal", async event => {
  if (!event.viewTransition)
    return;

  const x = sessionStorage.getItem("lastClickX") ?? innerWidth / 2;
  const y = sessionStorage.getItem("lastClickY") ?? innerHeight / 2;

  const endRadius = Math.hypot(
    Math.max(x, innerWidth - x),
    Math.max(y, innerHeight - y)
  );

  await event.viewTransition.ready;

  // Animate the new document's view
  document.documentElement.animate(
    {
      clipPath: [
        <code data-opaque bs-autolink-syntax="`circle(0 at ${x}px ${y}px)`">circle(0 at
        <code data-opaque bs-autolink-syntax="`circle(${endRadius}px at ${x}px ${y}px)`"
      ],
    },
    {
      duration: 500,
      easing: 'ease-in',
      pseudoElement: '::view-transition-new(root)'
    }
  );
})
```

EXAMPLE 5 ¶

To choose which elements are captured based on properties of the navigation, and whether certain images are loaded:

In the old page:

```
window.addEventListener("pageswap", event => {
  // For example, the page was hidden, or the navigation is cross-document.
  if (!event.viewTransition)
    return;

  // If you don't want view transition for back/forward navigations...
  if (event.activation.navigationType === "traverse") {
    event.viewTransition.skipTransition();
  }

  const newURL = new URL(event.activation.entry.url);
  if (newURL.pathname === "/details" && thumbnail.complete) {
    thumbnail.classList.add("transition-to-hero");

    // This will cleanup the state if the page is restored from BFCache.
    event.viewTransition.finished.then(() => {
      thumbnail.classList.remove("transition-to-hero");
    });
  }

});
```

In the new page:

```
window.addEventListener("pagereveal", event => {
  // For example, the page was hidden, the navigation is cross-document, or the transiti
  if (!event.viewTransition)
    return;

  const oldURL = new URL(navigation.activation.from.url);
  if (newURL.pathname === "/list") {
    event.viewTransition.types.add("coming-from-list");

    // This will show the thumbnail until the view transition is finished.
    if (!hero.complete) {
      setToThumbnail(hero);
      event.viewTransition.finished.then(() => {
        setToFullResolution(hero);
      })
    }
  }
});
```

§ 2.3. Opting in to cross-document view transitions

§ 2.3.1. The '*@view-transition*' rule

The '@view-transition' rule is used by a document to indicate that cross-document navigations should setup and activate a `ViewTransition`.

The '@view-transition' rule has the following syntax:

```
@view-transition {
  <declaration-list>
}
```

The '@view-transition' rule accepts the navigation and types descriptors.

NOTE: as per default behavior, the '@view-transition' rule can be nested inside a conditional group rule such as '@media' or '@supports'.

When the '@view-transition' rule changes for `Document` *document*, the user agent must update the opt-in state for outbound transitions given *document*.

NOTE: this needs to be cached in the boolean because the result needs to be read in parallel, when navigating.

§ 2.3.2. The navigation descriptor

Name:	*'navigation'*
For:	'@view-transition'
Value:	auto | none
Initial:	none

The '***navigation***' descriptor opts in to automatically starting a view transition when performing a navigation of a certain type. Must be present on both the old and new document.

'none'
: There will be no transition.

'auto'
: The transition will be enabled if the navigation is same-origin, without cross-origin redirects, and whose `NavigationType` is

- `traverse`, or
- `push` or `replace`, with user navigation involvement not equal to `"browser UI"`.

> NOTE: Navigations excluded from 'auto' are for example, navigating via the URL address bar or clicking a bookmark, as well as any form of user or script initiated `reload`.

This at-rule conforms with the forward-compatible parsing requirement of CSS; conformant parsers that don't understand these rules will ignore them without error. Any descriptors that are not recognized or implemented by a given user agent, or whose value does not match the grammars given here or in a future version of this specification, must be ignored in their entirety; they do not make the '@view-transition' rule invalid.

2.3.3. Accessing the '@view-transition' rule using CSSOM

The `CSSViewTransitionRule` represents a '@view-transition' rule.

```
[Exposed=Window]
interface CSSViewTransitionRule : CSSRule {
  readonly attribute CSSOMString navigation;
  [SameObject] readonly attribute FrozenArray<CSSOMString> types;
};
```

The `navigation` getter step is to return the value of the corresponding 'navigation' descriptor if one exists, otherwise the empty string.

The `types` getter steps is to return the value of the corresponding 'types' descriptor if one exists, otherwise an empty list.

3. Selective view transitions

3.1. Overview

This section is non-normative.

For simple pages, with a single view transition, setting the 'view-transition-name' property on participating elements should be sufficient. However, in more complex scenarios, the author might want to declare various view transitions, and only run one of them simultaneously. For example, sliding the whole page when clicking on the navigation bar, and sorting a list when one of its items is dragged.

To make sure each view transition only captures what it needs to, and different transitions don't interfere with each other, this spec introduces the concept of active types, alongside the ':active-view-transition' and ':active-view-transition-type()' pseudo-classes.

':active-view-transition' matches the document element when it has an active view transition, and ':active-view-transition-type()' matches the document element if the types in the selectors match the active view transition's active types.

The `ViewTransition`'s active types are populated in one of the following ways:

1. Passed as part of the arguments to `startViewTransition(callbackOptions)`
2. Mutated at any time, using the transition's `types`
3. Declared for a cross-document view transition, using the types descriptor.

§ 3.2. Examples

EXAMPLE 6 ¶

For example, the developer might start a transition in the following manner:

```
document.startViewTransition({update: updateTheDOMSomehow, types: ["slide-in", "reverse"
```

This will activate any of the following selectors:

```
:root:active-view-transition-type(slide-in) {}
:root:active-view-transition-type(reverse) {}
:root:active-view-transition-type(slide-in, reverse) {}
:root:active-view-transition-type(slide-in, something-else) {}
:root:active-view-transition {}
```

While starting a transition without providing transition types, would only activate ':active-view-transition":

```
document.startViewTransition(updateTheDOMSomehow);
// or
document.startViewTransition({update: updateTheDOMSomehow});


/* This would be active */
:root { }
:root:active-view-transition {}

/* This would not be active */
:root:active-view-transition-type(slide-in) {}
:root:active-view-transition-type(any-type-at-all-except-star) {}
```

§ 3.3. Selecting based on the active view transition

§ 3.3.1. The ':active-view-transition' pseudo-class

The *':active-view-transition'* pseudo-class applies to the root element of the document, if it has an active view transition.

The specificity of an ':active-view-transition' is one pseudo-class selector.

An ':active-view-transition' pseudo-class matches the document element when its node document has an non-null active view transition.

§ 3.3.2. The ':active-view-transition-type()' pseudo-class

The *':active-view-transition-type()'* pseudo-class applies to the root element of the document, if it has a matching active view transition. It has the following syntax definition:

`:active-view-transition-type(<custom-ident>#)`

The specificity of an ':active-view-transition-type()' is one pseudo-class selector.

An ':active-view-transition-type()' pseudo-class matches the document element when its node document has an non-null active view transition, whose active types contains at least one of the <custom-ident> arguments.

§ 3.4. Changing the types of an ongoing view transition

The `ViewTransition` interface is extended as follows:

```
[Exposed=Window]
interface ViewTransitionTypeSet {
  setlike<DOMString>;
};

[Exposed=Window]
partial interface ViewTransition {
  attribute ViewTransitionTypeSet types;
};
```

The `ViewTransitionTypeSet` object represents a set of strings, without special semantics.

NOTE: a `ViewTransitionTypeSet` can contain strings that are invalid for ':active-view-transition-type', e.g. strings that are not a <custom-ident>.

The `types` getter steps are to return this's active types.

§ 3.5. Activating the transition type for cross-document view transitions

The types descriptor

Name:	*'types'*
For:	'@view-transition'

Value:	none | <custom-ident>+
Initial:	none

The '***types***' descriptor sets the active types for the transition when capturing or performing the transition, equivalent to calling `startViewTransition(callbackOptions)` with that `types`.

NOTE: the types descriptor only applies to the `Document` in which it is defined. The author is responsible for using their chosen set of types in both documents.

§ 4. Sharing styles between view transition pseudo-elements

§ 4.1. Overview

This section is non-normative.

When styling multiple elements in the DOM in a similar way, it is common to use the class attribute: setting a name that's shared across multiple elements, and then using the class selector to declare the shared style.

The view transition pseudo-elements (e.g. 'view-transition-group()') are not defined in the DOM, but rather by using the 'view-transition-name' property. For that purpose, the 'view-transition-class" CSS property provides view transitions with the equivalent of HTML classes. When an element with a 'view-transition-name' also has a 'view-transition-class' value, that class would be selectable by the pseudo-elements, as per the examples.

§ 4.2. Examples

EXAMPLE 7 ¶

This example creates a transition with each box participating under its own name, while applying a 1-second duration to the animation of all the boxes:

```
<div class="box" id="red-box"></div>
<div class="box" id="green-box"></div>
<div class="box" id="yellow-box"></div>


div.box {
  view-transition-class: any-box;
  width: 100px;
  height: 100px;
}
#red-box {
  view-transition-name: red-box;
  background: red;
}
#green-box {
  view-transition-name: green-box;
  background: green;
}
#yellow-box {
  view-transition-name: yellow-box;
  background: yellow;
}

/* The following style would apply to all the boxes, thanks to 'view-transition-class' *
::view-transition-group(*.any-box) {
  animation-duration: 1s;
}
```

§ 4.3. The 'view-transition-class' property

Name:	***'view-transition-class'***
Value:	none | <custom-ident>+
Initial:	none
Applies to:	all elements
Inherited:	no
Percentages:	n/a
Computed value:	as specified
Canonical order:	per grammar
Animation type:	discrete

The 'view-transition-class' can be used to apply the same style rule to multiple named view transition pseudo-elements which may have a different 'view-transition-name'. While 'view-transition-name' is used to match between the element in

the old state with its corresponding element in the new state, 'view-transition-class' is used only to apply styles using the view transitionpseudo-elements ('::view-transition-group()', '::view-transition-image-pair()', '::view-transition-old()', '::view-transition-new()').

Note that 'view-transition-class' by itself doesn't mark an element for capturing, it is only used as an additional way to style an element that already has a 'view-transition-name'.

'none'
No class would apply to the named view transition pseudo-elements generated for this element.

'<custom-ident>+'
All of the specified <custom-ident> values (apart from 'none') are applied when used in named view transition pseudo-element selectors. 'none' is an invalid <custom-ident> for 'view-transition-class', even when combined with another <custom-ident>.

Each 'view transition class' is a tree-scoped name.

> NOTE: If the same 'view-transition-name' is specified for an element both in the old and new states of the transition, only the 'view-transition-class' values from the new state apply. This also applies for cross-document view transitions: classes from the old document would only apply if their corresponding 'view-transition-name' was not specified in the new document.

§ 4.4. Additions to named view transition pseudo-elements

The named view transition pseudo-elements ('view-transition-group()', 'view-transition-image-pair()', 'view-transition-old()', 'view-transition-new()') are extended to support the following syntax:

```
::view-transition-group(<pt-name-and-class-selector>)
::view-transition-image-pair(<pt-name-and-class-selector>)
::view-transition-old(<pt-name-and-class-selector>)
::view-transition-new(<pt-name-and-class-selector>)
```

where <pt-name-selector> works as previously defined, and <pt-name-and-class-selector> has the following syntax definition:

```
<pt-name-and-class-selector> = <pt-name-selector> <pt-class-selector>? | <pt-class-selector>
<pt-class-selector> = ['.' <custom-ident>]+
```

When interpreting the above grammar, white space is forbidden:

- Between <pt-name-selector> and <pt-class-selector>
- Between any of the components of <pt-class-selector>.

A named view transition pseudo-element selector which has one or more <custom-ident> values in its <pt-class-selector> would only match an element if the class list value in named elements for the pseudo-element's 'view-transition-name' contains all of those values.

The specificity of a named view transition pseudo-element selector with either:

- a <pt-name-selector> with a <custom-ident>; or
- a <pt-class-selector> with at least one <custom-ident>,

is equivalent to a type selector.

The specificity of a named view transition pseudo-element selector with a '*' argument and with an empty <pt-class-selector> is zero.

§ 5. Extending `document.startViewTransition()`

```
dictionary StartViewTransitionOptions {
  ViewTransitionUpdateCallback? update = null;
  sequence<DOMString>? types = null;
};

partial interface Document {
  ViewTransition startViewTransition(optional (ViewTransitionUpdateCallback or StartViewTr
};
```

The method steps for ***startViewTransition(*callbackOptions*)*** are as follows:

1. Let *updateCallback* be null.
2. If *callbackOptions* is a `ViewTransitionUpdateCallback`, set *updateCallback* to *callbackOptions*.
3. Otherwise, if *callbackOptions* is a `StartViewTransitionOptions`, then set *updateCallback* to *callbackOptions*'s `update`.
4. If *this*'s active view transition is not null and its outbound post-capture steps is not null, then:
   1. Let *preSkippedTransition* be a new `ViewTransition` in *this*'s relevant realm whose update callback is *updateCallback*.

      > NOTE: The *preSkippedTransition*'s `types` are ignored here because the transition is never activated.

   2. Skip *preSkippedTransition* with an "`InvalidStateError`" `DOMException`.
   3. Return *preSkippedTransition*.

> NOTE: This ensures that a same-document transition that started after firing `pageswap` is skipped.

5. Let *viewTransition* be the result of running the method steps for `startViewTransition(updateCallback)` given *updateCallback*.
6. If *callbackOptions* is a `StartViewTransitionOptions`, set *viewTransition*'s active types to a clone of `types` as a set.
7. Return *viewTransition*.

§ 6. Determining 'view-transition-name' automatically

§ 6.1. Overview

This section is non-normative.

For an element to participate in a view transition, it needs to receive a unique 'view-transition-name'. This can be tedious and verbose when multiple elements are involved in the same view transition, especially in cases where many of those are same-element transitions, as in, the element has the same 'view-transition-name' in the old and new state.

To make this easier, setting the 'view-transition-name' to 'auto' would generate a 'view-transition-name' for the element, or take it from the element's id if present.

§ 6.2. Examples

EXAMPLE 8 ¶

In this example, view transition is used to sort a list in an animated way:

```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
  <li>Item 4</li>
  ...
</ul>
```

Ordinarily, each of these items would have to receive a unique 'view-transition-name':

```
li:nth-child(1) { view-transition-name: item1; }
li:nth-child(2) { view-transition-name: item1; }
li:nth-child(3) { view-transition-name: item1; }
li:nth-child(4) { view-transition-name: item1; }
...
```

With 'auto', this CSS would work:

```
li {
  view-transition-name: auto;
}
```

§ 6.3. Additions to 'view-transition-name'

In addition to the existing values, the 'view-transition-name' also accepts an ***auto*** keyword. To resolve the used value of 'view-transition-name' for *element*:

1. Let *computed* be the computed value of 'view-transition-name'.
2. If *computed* is 'none', return null.
3. If *computed* is a <custom-ident>, return *computed*.
4. Assert: *computed* is 'auto'.
5. If *element* has an associated id, and *computed* is associated with the same root as *element*'s root, then return the value of *element*'s id.

   NOTE: this means that a '::part()' pseudo-element wouldn't resolve to its matching element's id.

6. Return a unique string. The string should remain consistent and unique for this element and `Document`, at least for the lifetime of *element*'s node document's active view transition.

   NOTE: this string is not web-observable, and is used for addressing the element in internal algorithms.

   NOTE: When used in a cross-document view transition, generated 'auto' values never match, resulting in separate '::view-transition-group()' pseudo-elements, one exiting and one entering.

A 'view-transition-name' generated by 'auto' is a tree-scoped name.

§ 7. Nested view-transitions

§ 7.1. Overview

This section is non-normative.

By default, setting `view-transition-name` on multiple elements generates a flat view transition tree, where all the '::view-transition-group()' pseudo-elements are children of the '::view-transition' pseudo-element. This is sufficient for

many simple use-cases, but there are some styling use-cases that cannot be achieved with a flat tree. The most prominent use-case is clipping: with a flat tree, all pseudo-elements are clipped to the snapshot containing block, so clipping elements in the normal tree would lose their clipping during the view-transition, resulting in a broken visual effect. The effects that have can have an unexpected visual effect in a flat tree:

- Clipping ('overflow', 'clip-path', 'border-radius'): clipping affects the children of the element.
- 'opacity', 'mask-image' and 'filter': These effects that are designed to work on a fully rasterized image of a tree, rather than on each item individually.
- 3D transforms ('transform-style', 'transform', 'perspective'): to display the full range of 3D transform animations, some hierarchy needs to be kept.

To enable these use cases, this specification introduces the concept of nesting view-transition pseudo-elements. By using the 'view-transition-group' CSS property, the author can assign a "parent group" for a generated '::view-transition-group()' pseudo-element, creating a hierarchy in the view transition tree.

§ 7.2. Examples

EXAMPLE 9 ¶

This example creates a transition where the view transition tree is nested instead of flat:

```
<section class="container">
  <article>Content</article>
</section>


.container {
  view-transition-name: container;
}

.container,
::view-transition-group(container) {
  clip-path: circle();
}

article {
  view-transition-name: article;
  view-transition-group: container;
}
```

By applying the 'clip-path' to both the containing element and its generated pseudo-element, we preserve the clip during the transition, and by applying 'view-transition-group' to the internal element referencing the container, we make the tree "nested" in a way that would apply this clipping.

§ 7.3. The 'view-transition-group' property

Name:	***'view-transition-group'***
Value:	normal | contain | nearest | <custom-ident>
Initial:	normal
Applies to:	all elements
Inherited:	no
Percentages:	n/a
Computed value:	as specified
Canonical order:	per grammar
Animation type:	discrete

The 'view-transition-group' property can be used in conjuction with 'view-transition-name' to generate a hierarchy of named view transition pseudo-element.

The used value for 'view-transition-group' resolves to a 'view-transition-name' in its ancestor chain, or to 'none'. When generating the named view transition pseudo-element, the '::view-transition-group()' with that name would be the parent of the '::view-transition-group()' generated for this element's 'view-transition-name'.

§ 8. Layered capture

§ 8.1. Overview

This section is non-normative.

In [css-view-transitions-1], the old and new states are captured as snapshots, and the initial and final geometry are captured, creating a crossfade animation by default. This is a simple model and mostly creates the desired outcome.

However, crossfading two flattened snapshots is not always the most expressive animation. CSS allows animating borders, gradient backgrounds, 'filter', 'box-shadow' etc. which can feel more expressive and natural than crossfade depending on

the desired UX.

In addition, when using nested view transitions, some of the animations could look "wrong" when the CSS properties are flattened to a snapshot. This includes tree effects, such as 'opacity', 'mask', 'clip-path' and 'filter', and also clipping using 'overflow'. These effects are designed to apply to the whole tree of elements, not just to one element and its content.

With layered capture, all the CSS properties that affect the entire tree, as well as box decorations, are captured as style and animate as part of the group.

ISSUE 1 should this behavior be an opt-in/opt-out with a CSS property? See issue 11078. ¶

§ 8.2. Table of captured CSS properties {#layered-captured-css-properties}

The following list of ***layered capture properties*** defines the CSS properties that participate in layered capture. When the old or new state of the element are captured, these properties are captured as style, and the user agent must render the snapshot with disregard to that property:

- 'background'
- 'border-left'
- 'border-top'
- 'border-bottom'
- 'border-right'
- 'border-radius'
- 'border-image'
- 'box-shadow'
- 'box-sizing'
- 'clip-path'
- 'filter'
- 'mask'
- 'opacity'
- 'outline'
- 'padding'

§ 9. Algorithms

§ 9.1. Data structures

§ 9.1.1. Additions to `Document`

A `Document` additionaly has:

inbound view transition params
: a view transition params, or null. Initially null.

can initiate outbound view transition
: a boolean. Initially false.

NOTE: this value can be read in parallel while navigating.

§ 9.1.2. Additions to `ViewTransition`

A `ViewTransition` additionally has:

active types
: A `ViewTransitionTypeSet`, initially empty.

outbound post-capture steps
: Null or a set of steps, initially null.

§ 9.1.3. Serializable view transition params

A ***view transition params*** is a struct whose purpose is to serialize view transition information across documents. It has the following items:

named elements
: a map, whose keys are strings and whose values are captured elements.

initial snapshot containing block size
: a tuple of two numbers (width and height).

§ 9.2. Additions to the view transition page-visibility change steps

The next steps are appended to the tasks view transition page-visibility change steps given *document*, after the current steps in the queued task:

1. Set *document*'s inbound view transition params to null.

NOTE: this is called from the HTML spec.

9.2.1. Captured elements extension

The captured element struct should contain these fields, in addition to the existing ones:

class list
: a list of strings, initially empty.

containing group name
: Null or a string, initially null.

old layered-capture style
: A string.

transform from snapshot containing block
: A matrix, initially the identity transform function.

old box properties
new box properties
: A layered box properties or null, initially null.

The ***layered box properties*** is a struct, containing the following fields:

box sizing
: 'border-box' or 'content-box'.

content box
padding box
: A rectangle, in CSS pixel units, relative to the border box.

9.3. Resolving the '@view-transition' rule

To ***resolve @view-transition rule*** for a `Document` *document*:

> NOTE: this is called in both the old and new document.

1. If *document*'s visibility state is "`hidden`", then return "`skip transition`".
2. Let *matchingRule* be the last '@view-transition' rule in *document*.
3. If *matchingRule* is not found, then return "`skip transition`".
4. If *matchingRule*'s navigation descriptor's computed value is 'none', then return "`skip transition`".
5. Assert: *matchingRule*'s navigation descriptor's computed value is 'auto'.
6. Let *typesDescriptor* be *matchingRule*'s types descriptor.
7. If *typesDescriptor*'s computed value is 'none', then return a set « ».
8. Return a set of strings corresponding to *typesDescriptor*'s computed value.

9.4. Setting up the view transition in the old `Document`

9.4.1. Check eligibility for outbound cross-document view transition

To check if a ***navigation can trigger a cross-document view-transition?*** given a `Document` *oldDocument*, a `Document` *newDocument*, a `NavigationType` *navigationType*, and a boolean *isBrowserUINavigation*:

> NOTE: this is called during navigation, potentially in parallel.

1. If the user agent decides to display an implementation-defined navigation experience, e.g. a gesture-based transition for a back navigation, the user agent may ignore the author-defined view transition. If that is the case, return false.
2. If *oldDocument*'s can initiate outbound view transition is false, then return false.
3. If *navigationType* is `reload`, then return false.
4. If *oldDocument*'s origin is not same origin as *newDocument*'s origin, then return false.
5. If *newDocument* was created via cross-origin redirects and *newDocument*'s latest entry is null, then return false.

   > NOTE: A Document's latest entry would be null if this is a new navigation, rather than a restore from BFCache.

6. If *navigationType* is `traverse`, then return true.
7. If *isBrowserUINavigation* is true, then return false.
8. Return true.

9.4.2. Setup the outbound transition when ready to swap pages

To ***setup cross-document view-transition*** given a `Document` *oldDocument*, a `Document` *newDocument*, and *proceedWithNavigation*, which is an algorithm accepting nothing:

> NOTE: This is called from the HTML spec.

1. Assert: These steps are running as part of a task queued on *oldDocument*.
2. If *oldDocument*'s can initiate outbound view transition is false, then return null.
3. Let *transitionTypesFromRule* be the result of resolving the @view-transition rule for *oldDocument*.
4. Assert: *transitionTypesFromRule* is not "`skip transition`".

NOTE: We don't know yet if *newDocument* has opted in, as it might not be parsed yet. We check the opt-in for *newDocument* when we fire the `pagereveal` event.

5. If *oldDocument*'s active view transition is not null, then skip *oldDocument*'s active view transition with an "`AbortError`" `DOMException` in *oldDocument*'s relevant Realm.

   NOTE: this means that any running transition would be skipped when the document is ready to unload.

6. Let *outboundTransition* be a new `ViewTransition` object in *oldDocument*'s relevant Realm.
7. Set *outboundTransition*'s active types to *transitionTypesFromRule*.

   NOTE: the active types are not shared between documents. Manipulating the `types` in the new document does not affect the types in *newDocument*, which would be read from the types descriptor once *newDocument* is revealed.

   NOTE: the `ViewTransition` is skipped once the old document is hidden.

8. Set *outboundTransition*'s outbound post-capture steps to the following steps given a view transition params-or-null *params*:

   1. Set *newDocument*'s inbound view transition params to *params*.

      NOTE: The inbound transition is activated after the dispatch of `pagereveal` to ensure mutations made in this event apply to the captured new state.

   2. To skip the transition after a timeout, the user agent may perform the following steps in parallel:

      1. Wait for an implementation-defined duration.
      2. Queue a global task on the DOM manipulation task source given *newDocument*'s relevant global object to perform the following step:

         1. If *newDocument*'s inbound view transition params is *params*, then set *newDocument*'s inbound view transition params to null.

   3. Call *proceedWithNavigation*.

9. Set *oldDocument*'s active view transition to *outboundTransition*.

   NOTE: The process continues in perform pending transition operations.

10. The user agent should display the currently displayed frame until either:

    - The `pagereveal` event is fired.
    - its active view transition's phase is "`done`".

    NOTE: this is to ensure that there are no unintended flashes between displaying the old and new state, to keep the transition smooth.

11. Return *outboundTransition*.

§ 9.4.3. Update the opt-in flag to reflect the current state

To ***update the opt-in state for outbound transitions*** for a `Document` *document*:

1. If *document* has been revealed, and the result of resolving the @view-transition rule is not "`skip transition`", then set *document*'s can initiate outbound view transition to true.
2. Otherwise, set *document*'s can initiate outbound view transition to false.

§ 9.4.4. Proceed with navigation if view transition is skipped

Append the following steps to skip the view transition given a `ViewTransition` *transition*:

1. If *transition*'s outbound post-capture steps is not null, then run *transition*'s outbound post-capture steps with null.

NOTE: This is written in a monkey-patch manner, and will be merged into the algorithm once the L1 spec graduates.

§ 9.4.5. Proceed with cross-document view transition after capturing the old state

Prepend the following step to the Perform pending transition operations algorithm given a `Document` *document*:

1. If *document*'s active view transition is not null and its outbound post-capture steps is not null, then:

   1. Assert: *document*'s active view transition's phase is "`pending-capture`".
   2. Let *viewTransitionParams* be null;
   3. Set *document*'s rendering suppression for view transitions to true.

      ISSUE 2 Though capture the old state appears here as a synchronous step, it is in fact an asynchronous step as rendering an element into an image cannot be done synchronously. This should be more explicit in the L1 spec. ¶

   4. Capture the old state for *transition*.

5. If this succeeded, then set *viewTransitionParams* to a new view transition params whose named elements is a clone of *transition*'s named elements, and whose initial snapshot containing block size is *transition*'s initial snapshot containing block size.
6. Set *document*'s rendering suppression for view transitions to false.
7. Call *transition*'s outbound post-capture steps given *viewTransitionParams*.

NOTE: This is written in a monkey-patch manner, and will be merged into the algorithm once the L1 spec graduates.

§ 9.5. Activating the view transition in the new `Document`

To ***resolve inbound cross-document view-transition*** for `Document` *document*:

1. Assert: *document* is fully active.
2. Assert: *document* has been revealed is true.
3. Update the opt-in state for outbound transitions for *document*.
4. Let *inboundViewTransitionParams* be *document*'s inbound view transition params.
5. If *inboundViewTransitionParams* is null, then return null.
6. Set *document*'s inbound view transition params to null.
7. If *document*'s active view transition is not null, then return null.

   NOTE: this means that starting a same-document transition before revealing the document would cancel a pending cross-document transition.
8. Resolve @view-transition rule for *document* and let *resolvedRule* be the result.
9. If *resolvedRule* is "`skip transition`", then return null.
10. Let *transition* be a new `ViewTransition` in *document*'s relevant Realm, whose named elements is *inboundViewTransitionParams*'s named elements, and initial snapshot containing block size is *inboundViewTransitionParams*'s initial snapshot containing block size.
11. Set *document*'s active view transition to *transition*.
12. Resolve *transition*'s update callback done promise with undefined.
13. Set *transition*'s phase to "`update-callback-called`".
14. Set *transition*'s active types to *resolvedRule*.
15. Return *transition*.

§ 9.6. Capturing the 'view-transition-class'

When capturing the old or new state for an element, perform the following steps given a captured element *capture* and an element *element*:

1. Set *capture*'s class list to the computed value of *element*'s 'view-transition-class', if it is associated with *element*'s node document.

NOTE: This is written in a monkey-patch manner, and will be merged into the algorithm once the L1 spec graduates.

§ 9.7. Capturing and applying 'view-transition-group'

§ 9.7.1. Resolving the 'view-transition-group' value

To get the ***document-scoped view transition group*** of an `Element` *element*, perform the following steps:

1. Let *computedGroup* be the computed value of element's 'view-transition-group' property.
2. If *computedGroup* is associated with *element*'s node document, return *computedGroup*.
3. Return 'normal'.

§ 9.7.2. Resolving the containing group name

To resolve the ***nearest containing group name*** of an `Element` *element*, perform the following steps given a `ViewTransition` *viewTransition*:

1. Assert: *element* participates in *viewTransition*.
2. Let *ancestorGroup* be *element*'s nearest flat tree ancestor who participates in *viewTransition* and whose document-scoped view transition group is not 'normal'.
3. If *ancestorGroup* exists, return *ancestorGroup*'s document-scoped view transition name.
4. Return 'none'.

9.7.3. Resolving the group name

To resolve the ***used group name*** of an `Element` *element*, perform the following steps given a `ViewTransition` *transition*:

1. Assert: *element* participates in *transition*.
2. Let *group* be *element*'s document-scoped view transition group.
3. Let *containingGroupName* be *element*'s nearest containing group name given *transition*.
4. Return the first matching statement, switching on *group*:

'normal'
'contain'
containingGroupName.

> NOTE: an element with 'contain' becomes the nearest containing group name for its descendants.

'nearest'
The document-scoped view transition name of the element's nearest flat tree ancestor which participates in the *transition*.

'<custom-ident>'
group if the element has a flat tree ancestor whose document-scoped view transition name is *group* and participates in *transition*; Otherwise *containingGroupName*.

9.7.4. Compute the old 'view-transition-group'

When capturing the old state for an element, perform the following steps given a captured element *capturedElement*, a `ViewTransition` *transition*, and an element *element*:

1. Set *capturedElement*'s containing group name to *element*'s used group name given *transition*.

9.7.5. Compute the new 'view-transition-group'

When capturing the new state for an element, perform the following steps given a captured element *capturedElement* a `ViewTransition` *transition*, and an element *element*:

1. Set *capturedElement*'s containing group name to *element*'s used group name given *transition*.

9.7.6. Reparent a '::view-transition-group()' to its specified containing group

When setting up the transition pseudo-element for a captured element *capturedElement*, given a *transitionName* and a *transition*:

1. Let *containingGroupName* be *capturedElement*'s containing group name.
2. If *containingGroupName* is not null, then:
    1. Let *groupContainerElement* be *transition*'s named elements[*containingGroupName*].
    2. Let *group* be the '::view-transition-group()', whose view transition name is set to *transitionName*.
    3. Let *parentGroup* be the '::view-transition-group()', whose view transition name is set to *containingGroupName*.
    4. Append *group* to *parentGroup*.
    5. When setting the animation keyframes given *transform*, adjust the nested group transform to *transform*, given *groupContainerElement*'s old transform, *groupContainerElement*'s old width, *groupContainerElement*'s old height, and *groupContainerElement*'s old box properties.

> NOTE: It is TBD how this is resolved when the old and new groups mismatch. See Issue 10631.

9.7.7. Adjust the group's 'transform' to be relative to its containing '::view-transition-group()'

When updating the style of the transition pseudo-element, perform the following steps before setting the group styles rule, given a captured element *capturedElement*, a *transform*, and a `ViewTransition` *transition*:

1. Set *capturedElement*'s transform from snapshot containing block to *transform*.
2. If *capturedElement*'s containing group name is not null, then:
    1. Let *groupContainerElement* be *transition*'s named elements[*capturedElement*'s containing group name.
    2. Let *containerRect* be snapshot containing block if *capturedElement* is the document element, otherwise, *capturedElement*'s border box.
    3. Adjust the nested group transform to *transform*, given *groupContainerElement*'s transform from snapshot containing block, *containerRect*'s width, *containerRect*'s height, and *groupContainerElement*'s new box properties.

9.8. Capturing layered CSS properties

9.8.1. Compute the layered-capture style

To compute the ***layered capture style*** of an `Element` *element*:

1. Let *propertiesToCapture* be a new list corresponding to the layered capture properties,
2. ISSUE 3 Specify the behavior of 'overflow' and containment. See issue 11079. ¶
3. Let *styles* be a « ».
4. For each *property* in *propertiesToCapture*, append the concatentation of « *property*, `":"`, *element*'s computed value of *property*, `";"` » to *styles*.
5. Return the concatentation of *styles*.

To compute the ***layered capture geometry*** of an `Element` *element*, return a new layered box properties, whose box sizing is *element*'s computed 'box-sizing', padding box is *element*'s padding box, and whose content box is *element*'s content box.

To compute the ***default group size*** given *element* return *element*'s content box's size if *element*'s computed 'box-sizing' is 'content-box', otherwise *element*'s border box's size.

§ 9.8.2. Capture the old layered properties

When capturing the old state for an element, perform the following steps given a captured element *capturedElement* and an element *element*:

1. Set *capturedElement*'s old layered-capture style to *element*'s layered capture style.
2. Set *capturedElement*'s old box properties to *element*'s layered capture geometry.
3. Set *capturedElement*'s (old width, old height to *element*'s default group size.

§ 9.8.3. Adjustment to image capture size

When capturing the image, the natural dimensions of the image will be based on the element's content box's size rather than on the border box.

§ 9.8.4. Render the snapshot with layered capture

When capturing an element into a snapshot, only the element contents are painted, without the element's effects and box decorations. Specifically, the element's 'background', 'border', 'border-image', 'box-shadow', and 'outline' are not painted, and its 'border-radius', 'clip-path', 'filter', 'mask', and 'opacity' are not applied.

§ 9.8.5. Adjust the nested group transform

To ***adjust the nested group transform*** given a matrix *transform*, a matrix *parentTransform*, *borderBoxWidth*, *borderBoxHeight*, and a layered box properties *boxProperties*, perform the following steps:

1. Multiply *transform* with the inverse matrix of *parentTransform*.
2. Let (left, top) be the border edge based on *boxProperties*'s padding box inside (*borderBoxWidth*, *borderBoxHeight*).
3. Translate *transform* by (-left, -top).

> NOTE: These operations ensure that the default position of this nested group would be equivalent to the original element's position, given that by default a nested ':view-transition-group' would be positioned relative to the parent's padding edge.

§ 9.8.6. Apply the old layered-capture properties to '::view-transition-group()' keyframes

When setting up the transition pseudo-element for a captured element *capturedElement*, given a *transitionName*:

1. Let *keyframesName* be the concatentation of « `-ua-view-transition-group-anim-`, *transitionName* »
2. Append *capturedElement*'s old layered-capture style to the constructed rule for *keyframesName*.

§ 9.8.7. Apply the new layered-capture properties to '::view-transition-group()'

When updating the style of the transition pseudo-element, perform the following steps before setting the group styles rule, given a *transitionName*, a *capturedElement*, *width*, *height*, and an `Element` *element*:

> NOTE: this might change and *width*, *height*.

1. Let *style* be *element*'s layered capture style.
2. Set *capturedElement*'s new box properties to *element*'s layered capture properties.
3. Set (*width*, *height*) to the *element*'s default group size.
4. Append the concatentation of « `"::view-transition-group("`, *transitionName*, `") {"`, *style* , `"}"` » to the constructed user-agent stylesheet.
5. Let (*oldContentWidth*, *oldContentHeight*) be (*capturedElement*'s old width, *capturedElement*'s old height) if *capturedElement*'s old box properties is null, otherwise *capturedElement*'s old box properties's content box's size.
6. Let (*newContentWidth*, *newContentHeight*) be (*width*, *height*) if new box properties is null, otherwise *capturedElement*'s new box properties's content box's size.

7. Append the next string (with replaced variables) to the user agent stylesheet:

```
@keyframes -ua-view-transition-content-geometry-transitionName {
  from {
    width: oldContentWidth;
    height: oldContentHeight;
  }
}

:root::view-transition-image-pair(transitionName) {
  position: relative;
  inset: unset;
  width: newContentWidth;
  height: newContentHeight;
  animation-name: -ua-view-transition-transitionName;
  animation-direction: inherit;
  animation-timing-function: inherit;
  animation-iteration-count: inherit;
  animation-duration: inherit;
}
```

NOTE: the ':view-transition-image-pair' pseudo-element is using 'relative' positioning so that the group's 'padding' will take effect.

§ Privacy Considerations

This specification introduces no new privacy considerations.

§ Security Considerations

To prevent cross-origin issues, at this point cross-document view transitions can only be enabled for same-origin navigations. As discussed in WICG/view-transitions#200, this still presents two potential threats:

1. The cross-origin isolated capability in both documents might be different. This can cause a situation where a `Document` that is cross-origin isolated can read image data from a document that is not cross-origin isolated. This is already mitigated in [[css-view-transitions-1#sec], as the same restriction applies for captured cross-origin iframes.
2. A same-origin navigation might still occur via a cross-origin redirect, e.g. `https://example.com` links to `https://auth-provider.com/` which redirects back to `https://example.com/loggedin`.

   This can cause a (minor) situation where the cross-origin party would redirect the user to an unexpected first-party URL, causing an unexpected transition and obfuscating that fact that there was a redirect. To mitigate this, currently view transitions are disabled for navigations if the `Document` was created via cross-origin redirects. Note that this check doesn't apply when the `Document` is being reactivated, as in that case the cross-origin redirect has already taken place.

   NOTE: this only applies to server-side redirects. A client-side redirect, e.g. using [^meta/http-equiv/refresh^], is equivalent to a new navigation.
3. This feature exposes more information to CSS, as so far CSS was not aware of anything navigation-related. This can raise concerns around safety 3rd-party CSS. However, as a general rule, 3rd-party stylesheets should come from trusted sources to begin with, as CSS can learn about the document or change it in many ways.

See Issue #8684 and WICG/view-transitions#200 for detailed discussion.

§ Appendix A. Changes

This appendix is *informative*.

§ Changes from 2024-05-16 Working Draft

- First pass at layered capture (#10585)
- Allow transitions when traversing into a document that was created using cross-origin redirects (#11063#11063)
- Clarify a few nesting details (#10780 and #10633)
- Allow `auto` as a keyword for 'view-transition-name' (#8320)
- Clarify timeout behavior for inbound view transition (#10800)
- When hiding the document, and inbound cross-document view transition should be skipped. (#9822)
- Rename UpdateCallback to something more specific.
- Clarify that CSSViewTransitionRule returns an empty string for invalid/missing navigation descriptor (#10654)
- Initial spec for nested view transitions (#10334)
- `view-transition-class` is a tree-scoped name (#10529)
- Stop extending CSSRule with obsolete pattern (See #10606)

§ Conformance

§ Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and RFC 2119 terminology. The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in the normative parts of this document are to be interpreted as described in RFC 2119. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. [RFC2119]

Examples in this specification are introduced with the words “for example” or are set apart from the normative text with `class="example"`, like this:

> EXAMPLE 10 ¶
>
> This is an example of an informative example.

Informative notes begin with the word “Note” and are set apart from the normative text with `class="note"`, like this:

> Note, this is an informative note.

Advisements are normative sections styled to evoke special attention and are set apart from other normative text with `<strong class="advisement">`, like this:

> **UAs MUST provide an accessible alternative.**

§ Conformance classes

Conformance to this specification is defined for three conformance classes:

style sheet
: A CSS style sheet.

renderer
: A UA that interprets the semantics of a style sheet and renders documents that use them.

authoring tool
: A UA that writes a style sheet.

A style sheet is conformant to this specification if all of its statements that use syntax defined in this module are valid according to the generic CSS grammar and the individual grammars of each feature defined in this module.

A renderer is conformant to this specification if, in addition to interpreting the style sheet as defined by the appropriate specifications, it supports all the features defined by this specification by parsing them correctly and rendering the document accordingly. However, the inability of a UA to correctly render a document due to limitations of the device does not make the UA non-conformant. (For example, a UA is not required to render color on a monochrome monitor.)

An authoring tool is conformant to this specification if it writes style sheets that are syntactically correct according to the generic CSS grammar and the individual grammars of each feature in this module, and meet all other conformance requirements of style sheets as described in this module.

§ Partial implementations

So that authors can exploit the forward-compatible parsing rules to assign fallback values, CSS renderers **must** treat as invalid (and ignore as appropriate) any at-rules, properties, property values, keywords, and other syntactic constructs for which they have no usable level of support. In particular, user agents **must not** selectively ignore unsupported component values and honor supported values in a single multi-value property declaration: if any value is considered invalid (as unsupported values must be), CSS requires that the entire declaration be ignored.

§ Implementations of Unstable and Proprietary Features

To avoid clashes with future stable CSS features, the CSSWG recommends following best practices for the implementation of unstable features and proprietary extensions to CSS.

§ Non-experimental implementations

Once a specification reaches the Candidate Recommendation stage, non-experimental implementations are possible, and implementors should release an unprefixed implementation of any CR-level feature they can demonstrate to be correctly implemented according to spec.

To establish and maintain the interoperability of CSS across implementations, the CSS Working Group requests that non-experimental CSS renderers submit an implementation report (and, if necessary, the testcases used for that implementation report) to the W3C before releasing an unprefixed implementation of any CSS features. Testcases submitted to W3C are subject to review and correction by the CSS Working Group.

Further information on submitting testcases and implementation reports can be found from on the CSS Working Group’s website at https://www.w3.org/Style/CSS/Test/. Questions should be directed to the public-css-testsuite@w3.org mailing list.

§ Index

§ Terms defined by this specification

§ Terms defined by reference

[CSS-BACKGROUNDS-3] defines the following terms:
- background
- border
- border-image

[CSS-BORDERS-4] defines the following terms:
- border-bottom
- border-left
- border-radius
- border-right
- border-top
- box-shadow

[CSS-BOX-4] defines the following terms:
- border box
- content box
- padding
- padding box
- padding edge

[CSS-CASCADE-5] defines the following terms:
- computed value
- used value

[CSS-COLOR-4] defines the following terms:
- opacity

[CSS-CONDITIONAL-3] defines the following terms:
- @media
- @supports
- conditional group rule

[CSS-CONTAIN-2] defines the following terms:
- element contents

[CSS-IMAGES-3] defines the following terms:
- natural dimension

[CSS-MASKING-1] defines the following terms:
- clip-path
- mask
- mask-image

[CSS-OVERFLOW-3] defines the following terms:
- overflow

[CSS-POSITION-3] defines the following terms:
- relative

[CSS-SCOPING-1] defines the following terms:
- flat tree
- tree-scoped name

[CSS-SHADOW-PARTS-1] defines the following terms:
- ::part()

[CSS-SIZING-3] defines the following terms:
- border-box
- box-sizing
- content-box

[CSS-SYNTAX-3] defines the following terms:
- <declaration-list>

[CSS-TRANSFORMS-1] defines the following terms:
- identity transform function
- transform

[CSS-TRANSFORMS-2] defines the following terms:
- perspective
- transform-style

[CSS-UI-4] defines the following terms:
- outline

[CSS-VALUES-4] defines the following terms:
- #
- +
- <custom-ident>
- ?
- |

[CSS-VIEW-TRANSITIONS-1] defines the following terms:

- ::view-transition
- ::view-transition-group()
- ::view-transition-image-pair()
- ::view-transition-new()
- ::view-transition-old()
- <pt-name-selector>
- ViewTransition
- ViewTransitionUpdateCallback
- activate view transition
- active view transition
- capture the new state
- capture the old state
- captured element
- capturing the image
- document-scoped view transition name
- finished
- group styles rule
- initial snapshot containing block size
- named elements
- named view transition pseudo-elements
- none
- old height
- old transform
- old width
- perform pending transition operations
- phase
- ready
- rendering suppression for view transitions
- skip the view transition
- skipTransition()
- snapshot containing block
- startViewTransition(updateCallback)
- update callback
- update callback done promise
- updateCallbackDone
- view transition name
- view transition page-visibility change steps
- view transition tree
- view-transition-name

[CSS22] defines the following terms:

- element

[CSSOM-1] defines the following terms:

- CSSOMString
- CSSRule

[DOM] defines the following terms:

- Document
- Element
- ancestor
- class
- document
- document element
- id
- node document
- origin
- root

[FILTER-EFFECTS-1] defines the following terms:

- filter

[GEOMETRY-1] defines the following terms:

- matrix
- multiply

[HR-TIME-3] defines the following terms:

- duration

[HTML] defines the following terms:

- HTMLLinkElement
- NavigationType
- PageSwapEvent
- activation
- blocking (for HTMLScriptElement)
- blocking (for HTMLStyleElement)
- cross-origin isolated capability
- dom manipulation task source
- expect
- fully active
- has been revealed
- in parallel
- latest entry
- pagereveal
- pageswap
- push
- queue a global task
- reactivate
- relevant global object
- relevant realm
- reload
- render-blocking mechanism
- rendering opportunity
- replace
- same origin
- sessionStorage
- task
- traverse
- viewTransition (for PageRevealEvent)
- viewTransition (for PageSwapEvent)
- visibility state
- was created via cross-origin redirects

[INFRA] defines the following terms:

- append
- assert
- clone (for list)
- clone (for map)
- concatenate
- contain
- implementation-defined
- item
- list
- map
- set
- struct
- tuple

[SELECTORS-3] defines the following terms:

- *

[SELECTORS-4] defines the following terms:

- class selector
- selector
- specificity
- type selector

[WEBIDL] defines the following terms:

- AbortError
- DOMException
- DOMString
- Exposed
- FrozenArray
- InvalidStateError
- SameObject
- getter steps
- method steps
- resolve
- sequence
- this

§ References

§ Normative References

[CSS-BACKGROUNDS-3]
Elika Etemad; Brad Kemper. *CSS Backgrounds and Borders Module Level 3*. 11 March 2024. CR. URL: https://www.w3.org/TR/css-backgrounds-3/

[CSS-BORDERS-4]
CSS Borders and Box Decorations Module Level 4. Editor's Draft. URL: https://drafts.csswg.org/css-borders-4/

[CSS-BOX-4]
Elika Etemad. *CSS Box Model Module Level 4*. 4 August 2024. WD. URL: https://www.w3.org/TR/css-box-4/

[CSS-CASCADE-5]
Elika Etemad; Miriam Suzanne; Tab Atkins Jr.. *CSS Cascading and Inheritance Level 5*. 13 January 2022. CR. URL: https://www.w3.org/TR/css-cascade-5/

[CSS-COLOR-4]
Chris Lilley; Tab Atkins Jr.; Lea Verou. *CSS Color Module Level 4*. 13 February 2024. CR. URL: https://www.w3.org/TR/css-color-4/

[CSS-CONTAIN-2]
Tab Atkins Jr.; Florian Rivoal; Vladimir Levin. *CSS Containment Module Level 2*. 17 September 2022. WD. URL: https://www.w3.org/TR/css-contain-2/

[CSS-IMAGES-3]
Tab Atkins Jr.; Elika Etemad; Lea Verou. *CSS Images Module Level 3*. 18 December 2023. CR. URL: https://www.w3.org/TR/css-images-3/

[CSS-MASKING-1]
Dirk Schulze; Brian Birtles; Tab Atkins Jr.. *CSS Masking Module Level 1*. 5 August 2021. CR. URL: https://www.w3.org/TR/css-masking-1/

[CSS-OVERFLOW-3]
Elika Etemad; Florian Rivoal. *CSS Overflow Module Level 3*. 29 March 2023. WD. URL: https://www.w3.org/TR/css-overflow-3/

[CSS-SCOPING-1]
Tab Atkins Jr.; Elika Etemad. *CSS Scoping Module Level 1*. 3 April 2014. WD. URL: https://www.w3.org/TR/css-scoping-1/

[CSS-SIZING-3]
Tab Atkins Jr.; Elika Etemad. *CSS Box Sizing Module Level 3*. 17 December 2021. WD. URL: https://www.w3.org/TR/css-sizing-3/

[CSS-SYNTAX-3]
Tab Atkins Jr.; Simon Sapin. *CSS Syntax Module Level 3*. 24 December 2021. CR. URL: https://www.w3.org/TR/css-syntax-3/

[CSS-TRANSFORMS-1]
Simon Fraser; et al. *CSS Transforms Module Level 1*. 14 February 2019. CR. URL: https://www.w3.org/TR/css-transforms-1/

[CSS-TRANSFORMS-2]
Tab Atkins Jr.; et al. *CSS Transforms Module Level 2*. 9 November 2021. WD. URL: https://www.w3.org/TR/css-transforms-2/

[CSS-UI-4]
Florian Rivoal. *CSS Basic User Interface Module Level 4*. 16 March 2021. WD. URL: https://www.w3.org/TR/css-ui-4/

[CSS-VALUES-4]
Tab Atkins Jr.; Elika Etemad. *CSS Values and Units Module Level 4*. 12 March 2024. WD. URL: https://www.w3.org/TR/css-values-4/

[CSS-VIEW-TRANSITIONS-1]
Tab Atkins Jr.; Jake Archibald; Khushal Sagar. *CSS View Transitions Module Level 1*. 28 March 2024. CR. URL: https://www.w3.org/TR/css-view-transitions-1/

[CSS22]
Bert Bos. *Cascading Style Sheets Level 2 Revision 2 (CSS 2.2) Specification*. 12 April 2016. WD. URL: https://www.w3.org/TR/CSS22/

[CSSOM-1]
Daniel Glazman; Emilio Cobos Álvarez. *CSS Object Model (CSSOM)*. 26 August 2021. WD. URL: https://www.w3.org/TR/cssom-1/

[DOM]
Anne van Kesteren. *DOM Standard*. Living Standard. URL: https://dom.spec.whatwg.org/

[FILTER-EFFECTS-1]
Dirk Schulze; Dean Jackson. *Filter Effects Module Level 1*. 18 December 2018. WD. URL: https://www.w3.org/TR/filter-effects-1/

[GEOMETRY-1]
Simon Pieters; Chris Harrelson. *Geometry Interfaces Module Level 1*. 4 December 2018. CR. URL: https://www.w3.org/TR/geometry-1/

[HR-TIME-3]
Yoav Weiss. *High Resolution Time*. 19 July 2023. WD. URL: https://www.w3.org/TR/hr-time-3/

[HTML]
Anne van Kesteren; et al. *HTML Standard*. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[INFRA]
Anne van Kesteren; Domenic Denicola. *Infra Standard*. Living Standard. URL: https://infra.spec.whatwg.org/

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://datatracker.ietf.org/doc/html/rfc2119

[SELECTORS-3]
Tantek Çelik; et al. *Selectors Level 3*. 6 November 2018. REC. URL: https://www.w3.org/TR/selectors-3/

[SELECTORS-4]
Elika Etemad; Tab Atkins Jr.. *Selectors Level 4*. 11 November 2022. WD. URL: https://www.w3.org/TR/selectors-4/

[WEBIDL]
Edgar Chen; Timothy Gu. *Web IDL Standard*. Living Standard. URL: https://webidl.spec.whatwg.org/

§ Informative References

[CSS-CONDITIONAL-3]
Chris Lilley; David Baron; Elika Etemad. *CSS Conditional Rules Module Level 3*. 15 August 2024. CR. URL: https://www.w3.org/TR/css-conditional-3/

[CSS-POSITION-3]
Elika Etemad; Tab Atkins Jr.. *CSS Positioned Layout Module Level 3*. 10 August 2024. WD. URL: https://www.w3.org/TR/css-position-3/

[CSS-SHADOW-PARTS-1]
Tab Atkins Jr.; Fergal Daly. *CSS Shadow Parts*. 15 November 2018. WD. URL: https://www.w3.org/TR/css-shadow-parts-1/

§ Property Index

Name	Value	Initial	Applies to	Inh.	%ages	Animation type	Canonical order	Computed value
'view-transition-class'	none | <custom-ident>+	none	all elements	no	n/a	discrete	per grammar	as specified
'view-transition-group'	normal | contain | nearest | <custom-ident>	normal	all elements	no	n/a	discrete	per grammar	as specified

§ '@view-transition' Descriptors

Name	Value	Initial
'navigation'	auto | none	none
'types'	none | <custom-ident>+	none

§ IDL Index

```
[Exposed=Window]
interface CSSViewTransitionRule : CSSRule {
  readonly attribute CSSOMString navigation;
  [SameObject] readonly attribute FrozenArray<CSSOMString> types;
};

[Exposed=Window]
interface ViewTransitionTypeSet {
  setlike<DOMString>;
};

[Exposed=Window]
partial interface ViewTransition {
  attribute ViewTransitionTypeSet types;
};

dictionary StartViewTransitionOptions {
  ViewTransitionUpdateCallback? update = null;
  sequence<DOMString>? types = null;
};

partial interface Document {
  ViewTransition startViewTransition(optional (ViewTransitionUpdateCallback or StartViewTr
};
```

§ Issues Index

ISSUE 1 should this behavior be an opt-in/opt-out with a CSS property? See issue 11078. ↵

ISSUE 2 Though capture the old state appears here as a synchronous step, it is in fact an asynchronous step as rendering an element into an image cannot be done synchronously. This should be more explicit in the L1 spec. ↵

ISSUE 3 Specify the behavior of 'overflow' and containment. See issue 11079. ↵