# CSS View Transitions Module Level 1

W3C Candidate Recommendation Snapshot, 5 September 2023

▼ More details about this document

**This version:**
https://www.w3.org/TR/2023/CR-css-view-transitions-1-20230905/

**Latest published version:**
https://www.w3.org/TR/css-view-transitions-1/

**Editor's Draft:**
https://drafts.csswg.org/css-view-transitions-1/

**History:**
https://www.w3.org/standards/history/css-view-transitions-1

**Implementation Report:**
https://wpt.fyi/results/css/css-view-transitions

**Feedback:**
CSSWG Issues Repository

**Editors:**
Tab Atkins-Bittner (Google)
Jake Archibald (Google)
Khushal Sagar (Google)

**Suggest an Edit for this Spec:**
GitHub Editor

## Abstract

This module defines the View Transition API, along with associated properties and pseudo-elements, which allows developers to create animated visual transitions representing changes in the document state.

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

## Table of Contents

## § 1. Introduction

*This section is non-normative.*

This specification introduces a DOM API and associated CSS features that allow developers to create animated visual transitions, called ***view transitions*** between different states of a document.

### § 1.1. Separating Visual Transitions from DOM Updates

Traditionally, creating a visual transition between two document states required a period where both states were present in the DOM at the same time. In fact, it usually involved creating a specific DOM structure that could represent both states. For example, if one element was “moving” between containers, that element often needed to exist outside of either container for the period of the transition, to avoid clipping from either container or their ancestor elements.

This extra in-between state often resulted in UX and accessibility issues, as the structure of the DOM was compromised for a purely-visual effect.

View Transitions avoid this troublesome in-between state by allowing the DOM to switch between states instantaneously, then performing a customizable visual transition between the two states in another layer, using a static visual capture of the old state, and a live capture of the new state. These captures are represented as a tree of pseudo-elements (detailed in § 3.2 View Transition Pseudo-elements), where the old visual state co-exists with the new state, allowing effects such as cross-fading while animating from the old to new size and position.

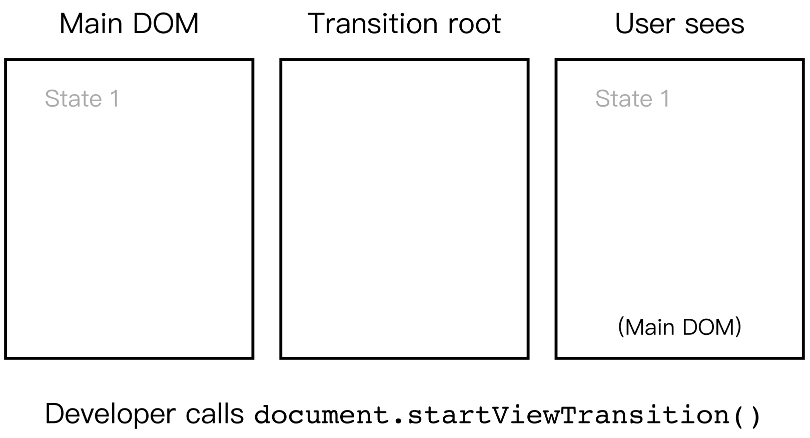
## § 1.2. View Transition Customization

By default, `document.startViewTransition()` creates a view transition consisting of a page-wide cross-fade between the two DOM states. Developers can also choose which elements are captured independently using the ‘view-transition-name’ CSS property, allowing these to be animated independently of the rest of the page. Since the transitional state (where both old and new visual captures exist) is represented as pseudo-elements, developers can customize each transition using familiar features such as CSS Animations and Web Animations.

## § 1.3. View Transition Lifecycle

A successful view transition goes through the following phases:

1. Developer calls `document.startViewTransition(updateCallback)`, which returns a `ViewTransition`, *viewTransition*.
2. Current state captured as the “old” state.
3. Rendering paused.
4. Developer’s `updateCallback` function is called, which updates the document state.
5. *`viewTransition`*`.updateCallbackDone` fulfills.
6. Current state captured as the “new” state.
7. Transition pseudo-elements created. See § 3.2 View Transition Pseudo-elements for an overview of this structure.
8. Rendering unpaused, revealing the transition pseudo-elements.
9. *`viewTransition`*`.ready` fulfills.
10. Pseudo-elements animate until finished.
11. Transition pseudo-elements removed.
12. *`viewTransition`*`.finished` fulfills.



## § 1.4. Transitions as an enhancement

A key part of the View Transition API design is that an animated transition is a visual *enhancement* to an underlying document state change. That means a failure to create a visual transition, which can happen due to misconfiguration or device constraints, will not prevent the developer’s `UpdateCallback` being called, even if it’s known in advance that the transition animations cannot happen.

For example, if the developer calls `skipTransition()` at the start of the view transition lifecycle, the steps relating to the animated transition, such as creating the view transition tree, will not happen. However, the `UpdateCallback` will still be called. It’s only the visual transition that’s skipped, not the underlying state change.

NOTE: If the DOM change should also be skipped, then that needs to be handled by another feature. `navigateEvent.signal` is an example of a feature developers could use to handle this.

Although the View Transition API allows DOM changes to be asynchronous via the `UpdateCallback`, the API is not responsible for queuing or otherwise scheduling DOM changes beyond any scheduling needed for the transition itself. Some asynchronous DOM changes can happen concurrently (e.g if they’re happening within independent components), whereas others need to queue, or abort an earlier change. This is best left to a feature or framework that has a more holistic view of the application.

## § 1.5. Rendering Model

View Transition works by replicating an element’s rendered state using UA generated pseudo-elements. Aspects of the element’s rendering which apply to the element itself or its descendants, for example visual effects like ‘filter’ or ‘opacity’ and clipping from ‘overflow’ or ‘clip-path’, are applied when generating its image in Capture the image.

However, properties like ‘mix-blend-mode’ which define how the element draws when it is embedded can’t be applied to its image. Such properties are applied to the element’s corresponding ‘::view-transition-group()’ pseudo-element, which is meant to generate a box equivalent to the element.

If the ‘::view-transition-group()’ has a corresponding element in the "new" states, the browser keeps the properties copied over to the ‘::view-transition-group()’ in sync with the DOM element in the "new" state. If the ‘::view-transition-group()’

has a corresponding both in the "old" and "new" state, and the property being copied is interpolatable, the browser also sets up a default animation to animate the property smoothly.

## § 1.6. Examples

EXAMPLE 1 ¶

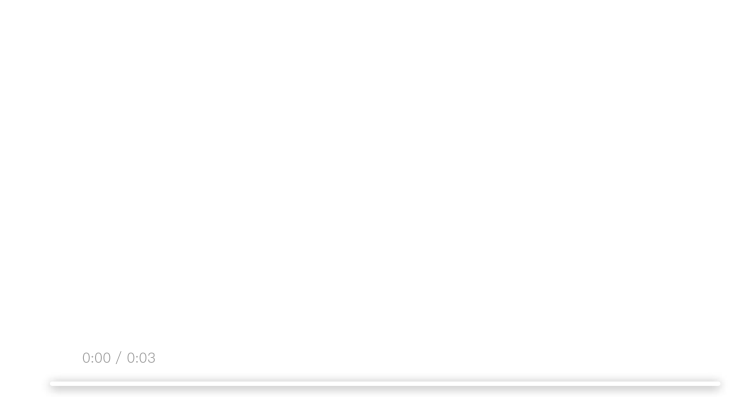
Taking a page that already updates its content using a pattern like this:

```
function spaNavigate(data) {
  updateTheDOMSomehow(data);
}
```

A view transition could be added like this:

```
function spaNavigate(data) {
  // Fallback for browsers that don’t support this API:
  if (!document.startViewTransition) {
    updateTheDOMSomehow(data);
    return;
  }

  // With a transition:
  document.startViewTransition(() => updateTheDOMSomehow(data));
}
```

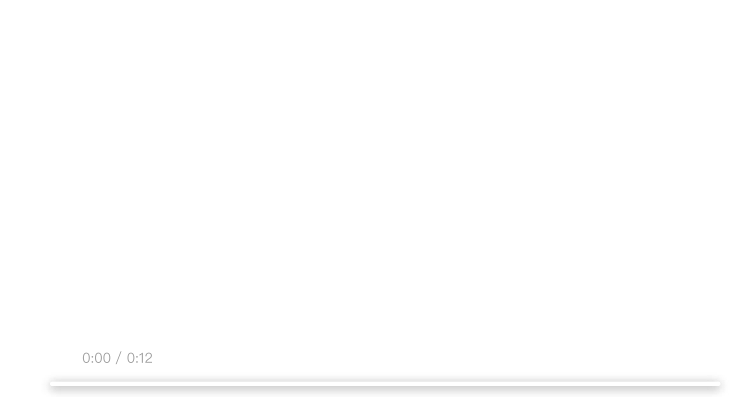
This results in the default transition of a quick cross-fade:



The cross-fade is achieved using CSS animations on a tree of pseudo-elements, so customizations can be made using CSS. For example:

```
::view-transition-old(root),
::view-transition-new(root) {
  animation-duration: 5s;
}
```

This results in a slower transition:

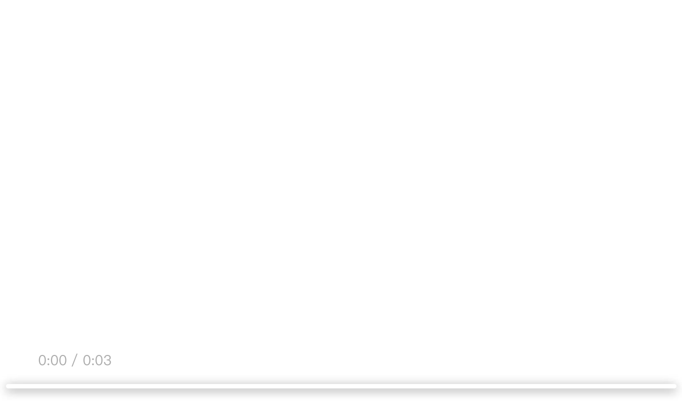


EXAMPLE 2 ¶

Building on the previous example, motion can be added:

```
@keyframes fade-in {
  from { opacity: 0; }
}

@keyframes fade-out {
  to { opacity: 0; }
}

@keyframes slide-from-right {
  from { transform: translateX(30px); }
}

@keyframes slide-to-left {
  to { transform: translateX(-30px); }
}

::view-transition-old(root) {
  animation: 90ms cubic-bezier(0.4, 0, 1, 1) both fade-out,
    300ms cubic-bezier(0.4, 0, 0.2, 1) both slide-to-left;
}

::view-transition-new(root) {
  animation: 210ms cubic-bezier(0, 0, 0.2, 1) 90ms both fade-in,
    300ms cubic-bezier(0.4, 0, 0.2, 1) both slide-from-right;
}
```

Here’s the result:

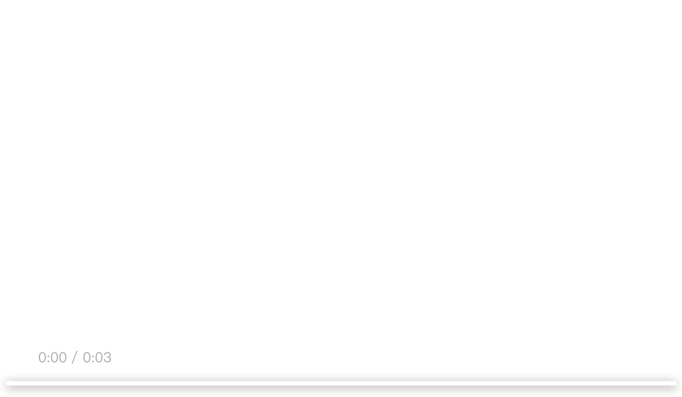


EXAMPLE 3 ¶

Building on the previous example, the header and text within the header can be given their own ‘::view-transition-group()’s for the transition:

```
.main-header {
  view-transition-name: main-header;
}

.main-header-text {
  view-transition-name: main-header-text;
  /* Give the element a consistent size, assuming identical text: */
  width: fit-content;
}
```

By default, these groups will transition size and position from their “old” to “new” state, while their visual states cross-fade:



EXAMPLE 4 ¶

Building on the previous example, let’s say some pages have a sidebar:

0:00 / 0:06

In this case, things would look better if the sidebar was static if it was in both the “old” and “new” states. Otherwise, it should animate in or out.

The ‘:only-child’ pseudo-class can be used to create animations specifically for these states:

```
.sidebar {
  view-transition-name: sidebar;
}

@keyframes slide-to-right {
  to { transform: translateX(30px); }
}

/* Entry transition */
::view-transition-new(sidebar):only-child {
  animation: 300ms cubic-bezier(0, 0, 0.2, 1) both fade-in,
    300ms cubic-bezier(0.4, 0, 0.2, 1) both slide-from-right;
}

/* Exit transition */
::view-transition-old(sidebar):only-child {
  animation: 150ms cubic-bezier(0.4, 0, 1, 1) both fade-out,
    300ms cubic-bezier(0.4, 0, 0.2, 1) both slide-to-right;
}
```

For cases where the sidebar has both an “old” and “new” state, the default animation is correct.

0:00 / 0:05

EXAMPLE 5

Not building from previous examples this time, let’s say we wanted to create a circular reveal from the user’s cursor. This can’t be done with CSS alone.

Firstly, in the CSS, allow the “old” and “new” states to layer on top of one another without the default blending, and prevent the default cross-fade animation:

```
::view-transition-image-pair(root) {
  isolation: auto;
}

::view-transition-old(root),
::view-transition-new(root) {
  animation: none;
  mix-blend-mode: normal;
}
```

Then, the JavaScript:

```
// Store the last click event
let lastClick;
addEventListener('click', event => (lastClick = event));

function spaNavigate(data) {
  // Fallback for browsers that don’t support this API:
  if (!document.startViewTransition) {
    updateTheDOMSomehow(data);
    return;
  }

  // Get the click position, or fallback to the middle of the screen
  const x = lastClick?.clientX ?? innerWidth / 2;
  const y = lastClick?.clientY ?? innerHeight / 2;
  // Get the distance to the furthest corner
  const endRadius = Math.hypot(
    Math.max(x, innerWidth - x),
    Math.max(y, innerHeight - y)
  );

  // Create a transition:
  const transition = document.startViewTransition(() => {
    updateTheDOMSomehow(data);
  });

  // Wait for the pseudo-elements to be created:
  transition.ready.then(() => {
    // Animate the root’s new view
    document.documentElement.animate(
      {
        clipPath: [
          `circle(0 at ${x}px ${y}px)`,
          `circle(${endRadius}px at ${x}px ${y}px)`,
        ],
      },
      {
        duration: 500,
        easing: 'ease-in',
        // Specify which pseudo-element to animate
        pseudoElement: '::view-transition-new(root)',
      }
    );
  });
}
```

And here’s the result:

0:00 / 0:04

## § 2. CSS properties

### § 2.1. Tagging Individually Transitioning Subtrees: the ‘view-transition-name’ property

| | |
|---|---|
| Name: | ‘view-transition-name’ |
| 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-name’ property “tags” an element for capture in a view transition, tracking it independently in the view transition tree under the specified ***view transition name***. An element so captured is animated independently of the rest of the page.

***‘none’***
The element will not participate independently in a view transition.

***‘<custom-ident>’***
The element participates independently in a view transition—as either an old or new element—with the specified view transition name.

The value ‘none’ is excluded from <custom-ident> here.

NOTE: If this name is not unique (i.e. if two elements simultaneously specify the same view transition name) then the view transition will abort.

NOTE: For the purposes of this API, if one element has view transition name ‘foo’ in the old state, and another element has view transition name ‘foo’ in the new state, they are treated as representing different visual state of the same element, and will be paired in the view transition tree. This may be confusing, since the elements themselves are not necessarily referring to the same object, but it is a useful model to consider them to be visual states of the same conceptual page entity.

If the element’s principal box is fragmented, skipped, or not rendered, this property has no effect. See § 7 Algorithms for exact details.

#### § 2.1.1. Rendering Consolidation

Elements captured in a view transition during a view transition or whose ‘view-transition-name’ computed value is not ‘none’ (at any time):

- Form a stacking context.
- Are flattened in 3D transforms.
- Form a backdrop root.

## § 3. Pseudo-elements

### § 3.1. Pseudo-element Trees

NOTE: This is a general definition for trees of pseudo-elements. If other features need this behavior, these definitions will be moved to [css-pseudo-4].

A ***pseudo-element root*** is a type of tree-abiding pseudo-element that is the root in a tree of tree-abiding pseudo-elements, known as the ***pseudo-element tree***.

The pseudo-element tree defines the document order of its descendant tree-abiding pseudo-elements.

When a pseudo-element participates in a pseudo-element tree, its originating pseudo-element is its parent.

If a descendant *pseudo* of a pseudo-element root has no other siblings, then ‘:only-child’ matches that *pseudo*.

NOTE: This means that `::view-transition-new(ident):only-child` will only select `::view-transition-new(ident)` if the parent `::view-transitions-image-pair(ident)` contains a single child. As in, there is no sibling `::view-transition-old(ident)`.

### § 3.2. View Transition Pseudo-elements

The visualization of a view transition is represented as a pseudo-element tree called the ***view transition tree*** composed of the ***view transition pseudo-elements*** defined below. This tree is built during the setup transition pseudo-elements step, and is rooted under a ‘::view-transition’ pseudo-element originating from the root element. All of the view transition pseudo-elements are selected from their ultimate originating element, the document element.

EXAMPLE 6 ¶
For example, the ‘::view-transition-group()’ pseudo-element is attached to the root element selector directly, as in ‘:root::view-transition-group()’; it is not attached to its parent, the ‘::view-transition’ pseudo-element.

Once the user-agent has captured both the “old” and “new” states of the document, it creates a structure of pseudo-elements like the following:

```
::view-transition
├─ ::view-transition-group(name)
│  └─ ::view-transition-image-pair(name)
│     ├─ ::view-transition-old(name)
│     └─ ::view-transition-new(name)
└─ …other groups…
```

Each element with a ‘view-transition-name’ is captured separately, and a ‘::view-transition-group()’ is created for each unique ‘view-transition-name’.

For convenience, the document element is given the ‘view-transition-name’ "root" in the user-agent style sheet.

Either ‘::view-transition-old()’ or ‘::view-transition-new()’ are absent in cases where the capture does not have an “old” or “new” state.

Each of the pseudo-elements generated can be targeted by CSS in order to customize its appearance, behavior and/or add animations. This enables full customization of the transition.

### § 3.2.1. Named View Transition Pseudo-elements

Several of the view transition pseudo-elements are ***named view transition pseudo-elements***, which are functional tree-abiding view transition pseudo-elements associated with a view transition name. These pseudo-elements take a <pt-name-selector> as their argument, and their syntax follows the pattern:

```
::view-transition-pseudo(<pt-name-selector>)
```

where <pt-name-selector> selects a view transition name, and has the following syntax definition:

```
<pt-name-selector> = '*' | <custom-ident>
```

A named view transition pseudo-element selector only matches a corresponding pseudo-element if its <pt-name-selector> matches that pseudo-element’s view transition name, i.e. if it is either ‘*’ or a matching <custom-ident>.

NOTE: The view transition name of a view transition pseudo-element is set to the ‘view-transition-name’ that triggered its creation.

The specificity of a named view transition pseudo-element selector with a <custom-ident> argument is equivalent to a type selector. The specificity of a named view transition pseudo-element selector with a ‘*’ argument is zero.

### § 3.2.2. View Transition Tree Root: the ‘::view-transition’ pseudo-element

The ***‘::view-transition’*** pseudo-element is a tree-abiding pseudo-element that is also a pseudo-element root. Its originating element is the document’s document element, and its containing block is the snapshot containing block.

NOTE: This element serves as the parent of all ‘::view-transition-group()’ pseudo-elements.

### § 3.2.3. View Transition Named Subtree Root: the ‘::view-transition-group()’ pseudo-element

The ***‘::view-transition-group()’*** pseudo-element is a named view transition pseudo-element that represents a matching named view transition capture. A ‘::view-transition-group()’ pseudo-element is generated for each view transition name as a child of the ‘::view-transition’ pseudo-element, and contains a corresponding ‘::view-transition-image-pair()’.

This element initially mirrors the size and position of the “old” element, or the “new” element if there isn’t an “old” element.
If there’s both an “old” and “new” state, styles in the dynamic view transition style sheet animate this pseudo-element’s ‘width’ and ‘height’ from the size of the old element’s border box to that of the new element’s border box.

Also the element’s ‘transform’ is animated from the old element’s screen space transform to the new element’s screen space transform.

This style is generated dynamically since the values of animated properties are determined at the time that the transition begins.

### § 3.2.4. View Transition Image Pair Isolation: the ‘::view-transition-image-pair()’ pseudo-element

The ***‘::view-transition-image-pair()’*** pseudo-element is a named view transition pseudo-element that represents a pair of corresponding old/new view transition captures. This pseudo-element is a child of the corresponding ‘::view-transition-group()’ pseudo-element and contains a corresponding ‘::view-transition-old()’ pseudo-element and/or a corresponding ‘::view-transition-new()’ pseudo-element (in that order).

This element exists to provide ‘isolation: isolate’ for its children, and is always present as a child of each ‘::view-transition-group()’. This isolation allows the image pair to be blended with non-normal blend modes without affecting other visual outputs.

#### § 3.2.5. View Transition Old State Image: the '::view-transition-old()' pseudo-element

The *'::view-transition-old()'* pseudo-element is an empty named view transition pseudo-element that represents a visual snapshot of the "old" state as a replaced element; it is omitted if there's no "old" state to represent. Each '::view-transition-old()' pseudo-element is a child of the corresponding '::view-transition-image-pair()' pseudo-element.

> ':only-child' can be used to match cases where this element is the only element in the '::view-transition-image-pair()'.
>
> The appearance of this element can be manipulated with `object-*` properties in the same way that other replaced elements can be.

> NOTE: The content and natural dimensions of the image are captured in capture the image, and set in setup transition pseudo-elements.

> NOTE: Additional styles in the dynamic view transition style sheet added to animate these pseudo-elements are detailed in setup transition pseudo-elements and update pseudo-element styles.

#### § 3.2.6. View Transition New State Image: the '::view-transition-new()' pseudo-element

The *'::view-transition-new()'* pseudo-element (like the analogous '::view-transition-old()' pseudo-element) is an empty named view transition pseudo-element that represents a visual snapshot of the "new" state as a replaced element; it is omitted if there's no "new" state to represent. Each '::view-transition-new()' pseudo-element is a child of the corresponding '::view-transition-image-pair()' pseudo-element.

> NOTE: The content and natural dimensions of the image are captured in capture the image, then set and updated in setup transition pseudo-elements and update pseudo-element styles.

## § 4. View Transition Layout

The view transition pseudo-elements are styled, laid out, and rendered like normal elements, except that they originate in the snapshot containing block rather than the initial containing block and are painted in the view transition layer above the rest of the document.

### § 4.1. The Snapshot Containing Block

The ***snapshot containing block*** is a rectangle that covers all areas of the window that could potentially display page content (and is therefore consistent regardless of root scrollbars or interactive widgets). This makes it likely to be consistent for the document element's old image and new element.

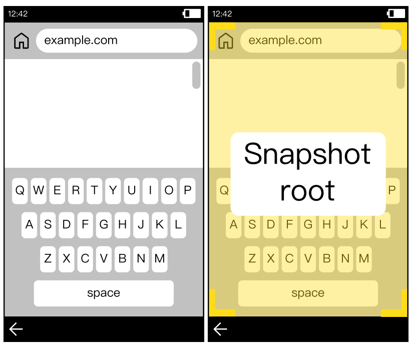


*Figure 1 An example of the snapshot containing block on a mobile OS. The snapshot includes the URL bar, as this can be scrolled away. The keyboard is included as this appears and disappears. The top and bottom bars are part of the OS rather than the browser, so they're not included in the snapshot containing block.*

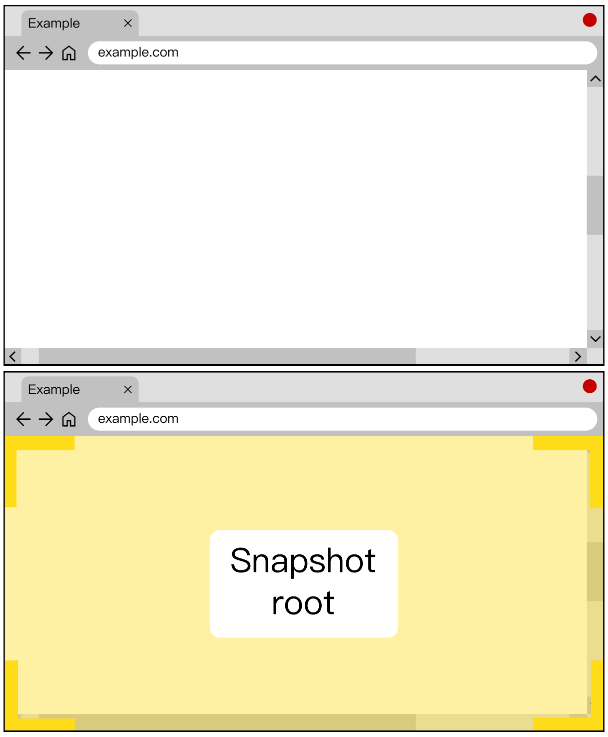


*Figure 2 An example of the snapshot containing block on a desktop OS. This includes the scrollbars, but does not include the URL bar, as web content never appears in that area.*

The ***snapshot containing block origin*** refers to the top-left corner of the snapshot containing block.

The ***snapshot containing block size*** refers to the width and height of the snapshot containing block as a tuple of two numbers.

The snapshot containing block is considered to be an absolute positioning containing block and a fixed positioning containing block for '::view-transition' and its descendants.

## § 4.2. View Transition Painting Order

This specification introduces a new stacking layer, the view transition layer, to the end of the painting order established in CSS2 § E Elaborate Description of Stacking Contexts. [CSS2]

The '::view-transition' pseudo-element generates a new stacking context, called the ***view transition layer***, which paints after all other content of the document (including any content rendered in the top layer), after any filters and effects that are applied to such content. (It is not subject to such filters or effects, except insofar as they affect the rendered contents of the '::view-transition-old()' and '::view-transition-new()' pseudo-elements.)

NOTE: The intent of the feature is to be able to capture the contents of the page, which includes the top layer elements. In order to accomplish that, the view transition layer cannot be a part of the captured stacking contexts, since that results in a circular dependency. Therefore, the view transition layer is a sibling of all other content.

When a `Document`'s active view transition's phase is "`animating`", the boxes generated by any element in that `Document` with captured in a view transition are invisible.

## § 5. User Agent Stylesheet

The ***global view transition user agent style sheet*** is a user-agent origin style sheet containing the following rules:

```
:root {
  view-transition-name: root;
}

:root::view-transition {
  position: fixed;
  inset: 0;
}

:root::view-transition-group(*) {
  position: absolute;
  top: 0;
  left: 0;

  animation-duration: 0.25s;
  animation-fill-mode: both;
}

:root::view-transition-image-pair(*) {
  position: absolute;
  inset: 0;
  animation-duration: inherit;
  animation-fill-mode: inherit;
}

:root::view-transition-old(*),
:root::view-transition-new(*) {
  position: absolute;
  inset-block-start: 0;
  inline-size: 100%;
  block-size: auto;

  animation-duration: inherit;
  animation-fill-mode: inherit;
}

/* Default cross-fade transition */
@keyframes -ua-view-transition-fade-out {
  to { opacity: 0; }
}
@keyframes -ua-view-transition-fade-in {
  from { opacity: 0; }
}

/* Keyframes for blending when there are 2 images */
@keyframes -ua-mix-blend-mode-plus-lighter {
  from { mix-blend-mode: plus-lighter }
  to { mix-blend-mode: plus-lighter }
}
```

▶ Explanatory Summary

Additional styles are dynamically added to the user-agent origin during a view transition through the dynamic view transition style sheet.

## § 6. API

### § 6.1. Additions to `Document`

```
partial interface Document {
  ViewTransition startViewTransition(optional UpdateCallback? updateCallback = null);
};

callback UpdateCallback = Promise<any> ();
```

For web developers (non-normative)

**`viewTransition = document.startViewTransition(updateCallback)`**

Starts a new view transition (canceling the `document`'s existing active view transition, if any).

`updateCallback` is called asynchronously, once the current state of the document is captured. Then, when the promise returned by `updateCallback` fulfills, the new state of the document is captured and the transition is initiated.

Note that `updateCallback` is *always* called, even if the transition cannot happen (e.g. due to duplicate `view-transition-name` values). The transition is an enhancement around the state change, so a failure to create a transition never prevents the state change. See § 1.4 Transitions as an enhancement for more details on this principle.

If the promise returned by `updateCallback` rejects, the transition is skipped.

#### § 6.1.1. `startViewTransition()` Method Steps

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

1. Let *transition* be a new `ViewTransition` object in this's relevant Realm.
2. Set *transition*'s update callback to *updateCallback*.
3. Let *document* be this's relevant global object's associated document.
4. If *document*'s active view transition is not null, then skip that view transition with an "`AbortError`" `DOMException` in this's relevant Realm.

   NOTE: This can result in two asynchronous update callbacks running concurrently (and therefore possibly out of sequence): one for the *document*'s current active view transition, and another for this *transition*. As per the design of this feature, it's assumed that the developer is using another feature or framework to correctly schedule these DOM changes.

5. Set *document*'s active view transition to *transition*.

   NOTE: The view transition process continues in setup view transition, via perform pending transition operations.

6. Return *transition*.

## § 6.2. The `ViewTransition` interface

```
[Exposed=Window]
interface ViewTransition {
  readonly attribute Promise<undefined> updateCallbackDone;
  readonly attribute Promise<undefined> ready;
  readonly attribute Promise<undefined> finished;
  undefined skipTransition();
};
```

The `ViewTransition` interface represents and controls a single same-document view transition, i.e. a transition where the starting and ending document are the same, possibly with changes to the document's DOM structure.

For web developers (non-normative)

**`viewTransition.updateCallbackDone`**

A promise that fulfills when the promise returned by `updateCallback` fulfills, or rejects when it rejects.

NOTE: The View Transition API wraps a DOM change and creates a visual transition. However, sometimes you don't care about the success/failure of the transition animation, you just want to know if and when the DOM change happens. `updateCallbackDone` is for that use-case.)

**`viewTransition.ready`**

A promise that fulfills once the pseudo-elements for the transition are created, and the animation is about to start.

It rejects if the transition cannot begin. This can be due to misconfiguration, such as duplicate 'view-transition-name's, or if `updateCallbackDone` returns a rejected promise.

The point that `ready` fulfills is the ideal opportunity to animate the view transition pseudo-elements with the Web Animation API.

**`viewTransition.finished`**

A promise that fulfills once the end state is fully visible and interactive to the user.

It only rejects if `updateCallback` returns a rejected promise, as this indicates the end state wasn't created.

Otherwise, if a transition fails to begin, or is skipped (by `skipTransition()`), the end state is still reached, so `finished` fulfills.

**`viewTransition.skipTransition()`**

Immediately finish the transition, or prevent it starting.

This never prevents `updateCallback` being called, as the DOM change is independent of the transition. See § 1.4 Transitions as an enhancement for more details on this principle.

If this is called before `ready` resolves, `ready` will reject.

If `finished` hasn't resolved, it will fulfill or reject along with `updateCallbackDone`.

A `ViewTransition` has the following:

***named elements***
a map, whose keys are view transition names and whose values are captured elements. Initially a new map.

***phase***
One of the following ordered phases:

1. `"pending-capture"`.
2. `"update-callback-called"`.
3. `"animating"`.
4. `"done"`.

NOTE: For the most part, a developer using this API does not need to worry about the different phases, since they progress automatically. It is, however, important to understand what steps happen in each of the phases: when the snapshots are captured, when pseudo-element DOM is created, etc. The description of the phases below tries to be as precise as possible, with an intent to provide an unambiguous set of steps for implementors to follow in order to produce a spec-compliant implementation.

***update callback***
an `UpdateCallback` or null. Initially null.

***ready promise***
a `Promise`. Initially a new promise in this's relevant Realm.

***update callback done promise***
a `Promise`. Initially a new promise in this's relevant Realm.

NOTE: The ready promise and update callback done promise are immediately created, so rejections will cause `unhandledrejection`s unless they're handled, even if the getters such as `updateCallbackDone` are not accessed.

***finished promise***
a `Promise`. Initially a new promise in this's relevant Realm, marked as handled.

NOTE: This is marked as handled to prevent duplicate `unhandledrejection`s, as this promise only ever rejects along with the update callback done promise.

***transition root pseudo-element***
a '::view-transition'. Initially a new '::view-transition'.

***initial snapshot containing block size***
a tuple of two numbers (width and height), or null. Initially null.

> NOTE: This is used to detect changes in the snapshot containing block size, which causes the transition to skip. Discussion of this behavior.

***process old state captured***
An algorithm accepting nothing, or null. Initially null.

> NOTE: this is used for cross-document view transitions.

A `ViewTransition` must never have both an update callback and a process old state captured.

> NOTE: update callback is optionally set for same-document view transitions, and process old state captured is set for cross-document view transitions.

The `finished` getter steps are to return this's finished promise.

The `ready` getter steps are to return this's ready promise.

The `updateCallbackDone` getter steps are to return this's update callback done promise.

#### § 6.2.1. `skipTransition()` Method Steps

The method steps for ***`skipTransition()`*** are:

1. If this's phase is not "`done`", then skip the view transition for this with an "`AbortError`" `DOMException`.

## § 7. Algorithms

### § 7.1. Data Structures

#### § 7.1.1. Additions to `Document`

A `Document` additionally has:

***active view transition***
a `ViewTransition` or null. Initially null.

***rendering suppression for view transitions***
a boolean. Initially false.

While a `Document`'s rendering suppression for view transitions is true, all pointer hit testing must target its document element, ignoring all other elements.

> NOTE: This does not affect pointers that are captured.

***dynamic view transition style sheet***
a style sheet. Initially a new style sheet in the user-agent origin, ordered after the global view transition user agent style sheet.

> NOTE: This is used to hold dynamic styles relating to transitions.

***show view transition tree***
A boolean. Initially false.

When this is true, this's active view transition's transition root pseudo-element renders as a child of this's document element, with this's document element is its originating element.

> NOTE: The position of the transition root pseudo-element within the document element does not matter, as the transition root pseudo-element's containing block is the snapshot containing block.

#### § 7.1.2. Additions to Elements

Elements have a ***captured in a view transition*** boolean, initially false.

> NOTE: This spec uses CSS's definition of element, which includes pseudo-elements.

#### § 7.1.3. Captured elements

A ***captured element*** is a struct with the following:

***old image***
an 2D bitmap or null. Initially null.

***old width***
***old height***
an `unrestricted double`, initially zero.

***old transform***
a ‘<transform-function>’, initially the identity transform function.

***old writing-mode***
Null or a ‘writing-mode’, initially null.

***old direction***
Null or a ‘direction’, initially null.

***old text-orientation***
Null or a ‘text-orientation’, initially null.

***old mix-blend-mode***
Null or a ‘mix-blend-mode’, initially null.

***new element***
an element or null. Initially null.

In addition, a captured element has the following ***style definitions***:

***group keyframes***
A `CSSKeyframesRule` or null. Initially null.

***group animation name rule***
A `CSSStyleRule` or null. Initially null.

***group styles rule***
A `CSSStyleRule` or null. Initially null.

***image pair isolation rule***
A `CSSStyleRule` or null. Initially null.

***image animation name rule***
A `CSSStyleRule` or null. Initially null.

NOTE: These are used to update, and later remove styles from a document's dynamic view transition style sheet.

## § 7.2. Perform pending transition operations

To ***perform pending transition operations*** given a `Document` *document*, perform the following steps:

1. If *document*’s active view transition is not null, then:
    1. If *document*’s active view transition's phase is "`pending-capture`", then setup view transition for *document*’s active view transition.
    2. Otherwise, if *document*’s active view transition's phase is "`animating`", then handle transition frame for *document*’s active view transition.

## § 7.3. Setup view transition

To ***setup view transition*** for a `ViewTransition` *transition*, perform the following steps:

NOTE: This algorithm captures the current state of the document, calls the transition’s `UpdateCallback`, then captures the new state of the document.

1. Let *document* be *transition*’s relevant global object’s associated document.
2. Capture the old state for *transition*.

    If failure is returned, then skip the view transition for *transition* with an "`InvalidStateError`" `DOMException` in *transition*’s relevant Realm, and return.
3. If *transition*’s process old state captured is not null, then call process old state captured and return.
4. Set *document*’s rendering suppression for view transitions to true.
5. Queue a global task on the DOM manipulation task source, given *transition*’s relevant global object, to execute the following steps:

    NOTE: A task is queued here because the texture read back in capturing the image may be async, although the render steps in the HTML spec act as if it’s synchronous.

    1. If *transition*’s phase is "`done`", then abort these steps.

        NOTE: This happens if *transition* was skipped before this point. The skip the view transition steps call the update callback, ensuring the *transition*’s update callback is always called.
    2. Call the update callback of *transition*.
    3. React to *transition*’s update callback done promise:
        - If the promise does not settle within an implementation-defined timeout, then:
            1. If *transition*’s phase is "`done`", then return.

                NOTE: This happens if *transition* was skipped before this point.
            2. Skip the view transition *transition* with a "`TimeoutError`" `DOMException`.
        - If the promise was rejected with reason *reason*, then:
            1. If *transition*’s phase is "`done`", then return.

                NOTE: This happens if *transition* was skipped before this point.
            2. Mark as handled *transition*’s ready promise.

                NOTE: *transition*’s update callback done promise will provide the `unhandledrejection`. This step avoids a duplicate.
            3. Skip the view transition *transition* with *reason*.

- If the promise was fulfilled, then activate *transition*.

To ***activate view transition*** for a `ViewTransition` *transition*, perform the following steps:

1. If *transition*'s phase is "`done`", then return.

   NOTE: This happens if *transition* was skipped before this point.

2. Set rendering suppression for view transitions to false.
3. If *transition*'s initial snapshot containing block size is not equal to the snapshot containing block size, then skip the view transition for *transition*, and return.
4. Capture the new state for *transition*.

   If failure is returned, then skip the view transition for *transition* with an "`InvalidStateError`" `DOMException` in *transition*'s relevant Realm, and return.
5. For each *capturedElement* of *transition*'s named elements' values:
   1. If *capturedElement*'s new element is not null, then set *capturedElement*'s new element's captured in a view transition to true.
6. Setup transition pseudo-elements for *transition*.
7. Update pseudo-element styles for *transition*.

   If failure is returned, then skip the view transition for *transition* with an "`InvalidStateError`" `DOMException` in *transition*'s relevant Realm, and return.

   NOTE: The above steps will require running document lifecycle phases, to compute information calculated during style/layout.
8. Set *transition*'s phase to "`animating`".
9. Resolve *transition*'s ready promise.

### § 7.3.1. Capture the old state

To ***capture the old state*** for `ViewTransition` *transition*:

1. Let *document* be *transition*'s relevant global object's associated document.
2. Let *namedElements* be *transition*'s named elements.
3. Let *usedTransitionNames* be a new set of strings.
4. Let *captureElements* be a new list of elements.
5. Let *document* be *transition*'s relevant global object's associated document.
6. Set *transition*'s initial snapshot containing block size to the snapshot containing block size.
7. For each *element* of every element that is connected, and has a node document equal to to *document*, in paint order:

   We iterate in paint order to ensure that this order is cached in *namedElements*. This defines the DOM order for ::view-transition-group pseudo-elements, such that the element at the bottom of the paint stack generates the first pseudo child of ::view-transition.

   1. If any flat tree ancestor of this *element* skips its contents, then continue.
   2. If *element* has more than one box fragment, then continue.

      NOTE: We might want to enable transitions for fragmented elements in future versions. See #8900.
   3. Let *transitionName* be the computed value of 'view-transition-name' for *element*.
   4. If *transitionName* is 'none', or *element* is not rendered, then continue.
   5. If *usedTransitionNames* contains *transitionName*, then return failure.
   6. Append *transitionName* to *usedTransitionNames*.
   7. Set *element*'s captured in a view transition to true.
   8. Append *element* to *captureElements*.

   The algorithm continues in a separate loop to ensure that captured in a view transition is set on all elements participating in this capture before it is read by future steps in the algorithm.
8. For each *element* in *captureElements*:
   1. Let *capture* be a new captured element struct.
   2. Set *capture*'s old image to the result of capturing the image of *element*.
   3. Let *originalRect* be snapshot containing block if *element* is the document element, otherwise, the element|'s border box.
   4. Set *capture*'s old width to *originalRect*'s `width`.
   5. Set *capture*'s old height to *originalRect*'s `height`.
   6. Set *capture*'s old transform to a '<transform-function>' that would map *element*'s border box from the snapshot containing block origin to its current visual position.
   7. Set *capture*'s old writing-mode to the computed value of 'writing-mode' on *element*.
   8. Set *capture*'s old direction to the computed value of 'direction' on *element*.
   9. Set *capture*'s old text-orientation to the computed value of 'text-orientation' on *element*.
   10. Set *capture*'s old mix-blend-mode to the computed value of 'mix-blend-mode' on *element*.

    11. Let *transitionName* be the computed value of ‘view-transition-name’ for *element*.
    12. Set *namedElements*[*transitionName*] to *capture*.
9. For each *element* in *captureElements*:
    1. Set *element*’s captured in a view transition to false.

### § 7.3.2. Capture the new state

To ***capture the new state*** for `ViewTransition` *transition*:

1. Let *document* be *transition*’s relevant global object’s associated document.
2. Let *namedElements* be *transition*’s named elements.
3. Let *usedTransitionNames* be a new set of strings.
4. For each *element* of every element that is connected, and has a node document equal to to *document*, in paint order:
    1. If any flat tree ancestor of this *element* skips its contents, then continue.
    2. Let *transitionName* be the computed value of ‘view-transition-name’ for *element*.
    3. If *transitionName* is ‘none’, or *element* is not rendered, then continue.
    4. If *usedTransitionNames* contains *transitionName*, then return failure.
    5. Append *transitionName* to *usedTransitionNames*.
    6. If *namedElements*[*transitionName*] does not exist, then set *namedElements*[*transitionName*] to a new captured element struct.
    7. Set *namedElements*[*transitionName*]'s new element to *element*.

### § 7.3.3. Setup transition pseudo-elements

To ***setup transition pseudo-elements*** for a `ViewTransition` *transition*:

NOTE: This algorithm constructs the pseudo-element tree for the transition, and generates initial styles. The structure of the pseudo-tree is covered at a higher level in § 3.2 View Transition Pseudo-elements.

1. Let *document* be this’s relevant global object’s associated document.
2. Set *document*’s show view transition tree to true.
3. For each *transitionName* → *capturedElement* of *transition*’s named elements:
    1. Let *group* be a new ‘::view-transition-group()’, with its view transition name set to *transitionName*.
    2. Append *group* to *transition*’s transition root pseudo-element.
    3. Let *imagePair* be a new ‘::view-transition-image-pair()’, with its view transition name set to *transitionName*.
    4. Append *imagePair* to *group*.
    5. If *capturedElement*’s old image is not null, then:
        1. Let *old* be a new ‘::view-transition-old()’, with its view transition name set to *transitionName*, displaying *capturedElement*’s old image as its replaced content.
        2. Append *old* to *imagePair*.
    6. If *capturedElement*’s new element is not null, then:
        1. Let *new* be a new ‘::view-transition-new()’, with its view transition name set to *transitionName*.

            NOTE: The styling of this pseudo is handled in update pseudo-element styles.
        2. Append *new* to *imagePair*.
    7. If *capturedElement*’s old image is null, then:
        1. Assert: *capturedElement*’s new element is not null.
        2. Set *capturedElement*’s image animation name rule to a new `CSSStyleRule` representing the following CSS, and append it to *document*’s dynamic view transition style sheet:

            ```
            :root::view-transition-new(transitionName) {
              animation-name: -ua-view-transition-fade-in;
            }
            ```

            NOTE: The above code example contains variables to be replaced.
    8. If *capturedElement*’s new element is null, then:
        1. Assert: *capturedElement*’s old image is not null.
        2. Set *capturedElement*’s image animation name rule to a new `CSSStyleRule` representing the following CSS, and append it to *document*’s dynamic view transition style sheet:

            ```
            :root::view-transition-old(transitionName) {
              animation-name: -ua-view-transition-fade-out;
            }
            ```

            NOTE: The above code example contains variables to be replaced.

9. If both of *capturedElement*'s old image and new element are not null, then:

    1. Let *transform* be *capturedElement*'s old transform.
    2. Let *width* be *capturedElement*'s old width.
    3. Let *height* be *capturedElement*'s old height.
    4. Set *capturedElement*'s group keyframes to a new `CSSKeyframesRule` representing the following CSS, and append it to *document*'s dynamic view transition style sheet:

```
@keyframes -ua-view-transition-group-anim-transitionName {
  from {
    transform: transform;
    width: width;
    height: height;
  }
}
```

    NOTE: The above code example contains variables to be replaced.

    5. Set *capturedElement*'s group animation name rule to a new `CSSStyleRule` representing the following CSS, and append it to *document*'s dynamic view transition style sheet:

```
:root::view-transition-group(transitionName) {
  animation-name: -ua-view-transition-group-anim-transitionName;
}
```

    NOTE: The above code example contains variables to be replaced.

    6. Set *capturedElement*'s image pair isolation rule to a new `CSSStyleRule` representing the following CSS, and append it to *document*'s dynamic view transition style sheet:

```
:root::view-transition-image-pair(transitionName) {
  isolation: isolate;
}
```

    NOTE: The above code example contains variables to be replaced.

    7. Set *capturedElement*'s image animation name rule to a new `CSSStyleRule` representing the following CSS, and append it to *document*'s dynamic view transition style sheet:

```
:root::view-transition-old(transitionName) {
  animation-name: -ua-view-transition-fade-out, -ua-mix-blend-mode-plus-lighte
}
:root::view-transition-new(transitionName) {
  animation-name: -ua-view-transition-fade-in, -ua-mix-blend-mode-plus-lighter
}
```

    NOTE: The above code example contains variables to be replaced.

    NOTE: 'mix-blend-mode: plus-lighter' ensures that the blending of identical pixels from the old and new images results in the same color value as those pixels, and achieves a "correct" cross-fade.

### § 7.4. Call the update callback

To ***call the update callback*** of a `ViewTransition` *transition*:

NOTE: This is guaranteed to happen for every `ViewTransition`, even if the transition is skipped. The reasons for this are discussed in § 1.4 Transitions as an enhancement.

1. Assert: *transition*'s phase is `"done"`, or before `"update-callback-called"`.
2. Let *callbackPromise* be null.
3. If *transition*'s update callback is null, then set *callbackPromise* to a promise resolved with undefined, in *transition*'s relevant Realm.
4. Otherwise, set *callbackPromise* to the result of invoking *transition*'s update callback.
5. If *transition*'s phase is not `"done"`, then set *transition*'s phase to `"update-callback-called"`.
6. Resolve *transition*'s update callback done promise with the result of reacting to *callbackPromise*:
    - If the promise was fulfilled, then return undefined.

    NOTE: Since the rejection of *callbackPromise* isn't explicitly handled here, if *callbackPromise* rejects, then *transition*'s update callback done promise will reject with the same reason.

### § 7.5. Skip the view transition

To ***skip the view transition*** for `ViewTransition` *transition* with reason *reason*:

1. Let *document* be *transition*'s relevant global object's associated document.

2. Assert: *document*'s active view transition is *transition*.
3. Assert: *transition*'s phase is not "`done`".
4. If *transition*'s phase is before "`update-callback-called`", then queue a global task on the DOM manipulation task source, given *transition*'s relevant global object, to call the update callback of *transition*.
5. Set rendering suppression for view transitions to false.
6. Clear view transition *transition*.
7. Set *transition*'s phase to "`done`".
8. Reject *transition*'s ready promise with *reason*.

   NOTE: The ready promise may already be resolved at this point, if `skipTransition()` is called after we start animating. In that case, this step is a no-op.

9. Resolve *transition*'s finished promise with the result of reacting to *transition*'s update callback done promise:
   - If the promise was fulfilled, then return undefined.

   NOTE: Since the rejection of *transition*'s update callback done promise isn't explicitly handled here, if *transition*'s update callback done promise rejects, then *transition*'s finished promise will reject with the same reason.

10. If *transition*'s process old state captured is not null, then call *transition*'s process old state captured.

## § 7.6. Capture the image

To ***capture the image*** given an element *element*, perform the following steps. They return an image.

1. If *element* is the document element, then:
   1. Render the region of document (including its canvas background and any top layer content) that intersects the snapshot containing block, on a transparent canvas the size of the snapshot containing block, following the capture rendering characteristics, and these additional characteristics:
      - Areas outside *element*'s scrolling box should be rendered as if they were scrolled to, without moving or resizing the layout viewport. This must not trigger events related to scrolling or resizing, such as `IntersectionObserver`s.

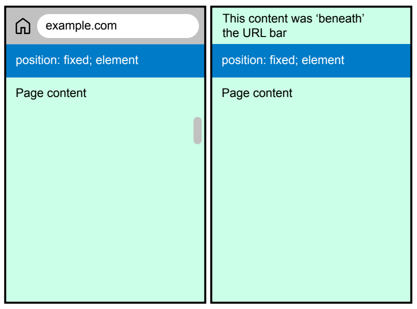


*Figure 3* *An example of what the user sees compared to the captured snapshot. This example assumes the root is the only element with a transition name.*

      - Areas that cannot be scrolled to (i.e. they are out of scrolling bounds), should render the canvas background.

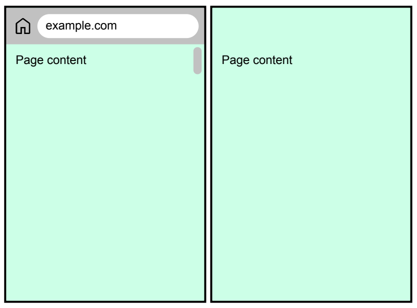


*Figure 4* *An example of what the user sees compared to the captured snapshot. This example assumes the root is the only element with a transition name.*

   2. Return this canvas as an image. The natural size of the image is equal to the snapshot containing block.
2. Otherwise:
   1. Render *element* and its descendants, at the same size it appears in its node document, over an infinite transparent canvas, following the capture rendering characteristics.
   2. Return the portion of this canvas that includes *element*'s ink overflow rectangle as an image. The natural dimensions of this image must be those of its principal border box, and its origin must correspond to that border box's origin, such that the image represents the contents of this border box and any captured ink overflow is represented outside these bounds.

NOTE: When this image is rendered as a replaced element at its natural size, it will display with the size and contents of element's principal box, with any captured ink overflow overflowing its content box.

#### § 7.6.1. Capture rendering characteristics

The ***capture rendering characteristics*** are as follows:

- If the referenced element has a transform applied to it (or its ancestors), then the transform is ignored.

  NOTE: This transform is applied to the snapshot using the `transform` property of the associated '::view-transition-group' pseudo-element.

- Effects applied on the element and its descendants, such as 'opacity' and 'filter', are applied to the capture. Effects applied to the element from its ancestors are ignored.
- Implementations may clip the rendered contents if the ink overflow rectangle exceeds some implementation-defined maximum. However, the captured image should include, at the very least, the contents of *element* that intersect with the snapshot containing block. Implementations may adjust the rasterization quality to account for elements with a large ink overflow area that are transformed into view.
- For each *descendant* of shadow-including descendant `Element` and pseudo-element of *element*, if *descendant* is captured in a view transition, then skip painting *descendant*.

  NOTE: This is necessary since the descendant will generate its own snapshot which will be displayed and animated independently.

### § 7.7. Handle transition frame

To ***handle transition frame*** given a `ViewTransition` *transition*:

1. Let *document* be *transition*'s relevant global object's associated document.
2. Let *hasActiveAnimations* be a boolean, initially false.
3. For each *element* of *transition*'s transition root pseudo-element's inclusive descendants:
    1. For each *animation* whose timeline is a document timeline associated with *document*, and contains at least one associated effect whose effect target is *element*, set *hasActiveAnimations* to true if any of the following conditions is true:
        - *animation*'s play state is paused or running.
        - *document*'s pending animation event queue has any events associated with *animation*.
4. If *hasActiveAnimations* is false:
    1. Set *transition*'s phase to "`done`".
    2. Clear view transition *transition*.
    3. Resolve *transition*'s finished promise.
    4. Return.
5. If *transition*'s initial snapshot containing block size is not equal to the snapshot containing block size, then skip the view transition for *transition*, and return.
6. Update pseudo-element styles for *transition*.

   If failure is returned, then skip the view transition for *transition* with an "`InvalidStateError`" `DOMException` in *transition*'s relevant Realm, and return.

   NOTE: The above implies that a change in incoming element's size or position will cause a new keyframe to be generated. This can cause a visual jump. We could retarget smoothly but don't have a use-case to justify the complexity. See issue 7813 for details.

### § 7.8. Update pseudo-element styles

To ***update pseudo-element styles*** for a `ViewTransition` *transition*:

1. For each *transitionName* → *capturedElement* of *transition*'s named elements:
    1. Let *width*, *height*, *transform*, *writingMode*, *direction*, *textOrientation* and *mixBlendMode* be null.
    2. If *capturedElement*'s new element is null, then:
        1. Set *width* to *capturedElement*'s old width.
        2. Set *height* to *capturedElement*'s old height.
        3. Set *transform* to *capturedElement*'s old transform.
        4. Set *writingMode* to *capturedElement*'s old writing-mode.
        5. Set *direction* to *capturedElement*'s old direction.
        6. Set *textOrientation* to *capturedElement*'s old text-orientation.
        7. Set *mixBlendMode* to *capturedElement*'s old mix-blend-mode.
    3. Otherwise:
        1. Return failure if any of the following conditions is true:

- *capturedElement*'s new element has a flat tree ancestor that skips its contents.
- *capturedElement*'s new element is not rendered.
- *capturedElement* has more than one box fragment.

NOTE: Other rendering constraints are enforced via *capturedElement*'s new element being captured in a view transition.

2. Set *width* to the current width of *capturedElement*'s new element's border box.
3. Set *height* to the current height of *capturedElement*'s new element's border box.
4. Set *transform* to a transform that would map *capturedElement*'s new element's border box from the snapshot containing block origin to its current visual position.
5. Set *writingMode* to the computed value of 'writing-mode' on *capturedElement*'s new element.
6. Set *direction* to the computed value of 'direction' on *capturedElement*'s new element.
7. Set *textOrientation* to the computed value of 'text-orientation' on *capturedElement*'s new element.
8. Set *mixBlendMode* to the computed value of 'mix-blend-mode' on *capturedElement*'s new element.

4. If *capturedElement*'s group styles rule is null, then set *capturedElement*'s group styles rule to a new `CSSStyleRule` representing the following CSS, and append it to *transition*'s relevant global object's associated document's dynamic view transition style sheet.

Otherwise, update *capturedElement*'s group styles rule to match the following CSS:

```
:root::view-transition-group(transitionName) {
  width: width;
  height: height;
  transform: transform;
  writing-mode: writingMode;
  direction: direction;
  text-orientation: textOrientation;
  mix-blend-mode: mixBlendMode;
}
```

NOTE: The above code example contains variables to be replaced.

5. If *capturedElement*'s new element is not null, then:
   1. Let *new* be the '::view-transition-new()' with the view transition name *transitionName*.
   2. Set *new*'s replaced element content to the result of capturing the image of *capturedElement*'s new element.

This algorithm must be executed to update styles in user-agent origin if its effects can be observed by a web API.

NOTE: An example of such a web API is `window.getComputedStyle(document.documentElement, "::view-transition")`.

## § 7.9. Clear view transition

To ***clear view transition*** of a `ViewTransition` *transition*:

1. Let *document* be *transition*'s relevant global object's associated document.
2. Assert: *document*'s active view transition is *transition*.
3. For each *capturedElement* of *transition*'s named elements' values:
   1. If *capturedElement*'s new element is not null, then set *capturedElement*'s new element's captured in a view transition to false.
   2. For each *style* of *capturedElement*'s style definitions:
      1. If *style* is not null, and *style* is in *document*'s dynamic view transition style sheet, then remove *style* from *document*'s dynamic view transition style sheet.
4. Set *document*'s show view transition tree to false.
5. Set *document*'s active view transition to null.

## § Privacy Considerations

This specification introduces no new privacy considerations.

## § Security Considerations

The images generated using capture the image algorithm could contain cross-origin data (if the Document is embedding cross-origin resources) or sensitive information like visited links. The implementations must ensure this data can not be accessed by the Document. This should be feasible since access to this data should already be prevented in the default rendering of the Document.

## § Appendix A. Changes

This appendix is *informative*.

### § Changes from 2022-05-30 Working Draft

- Use a keyframe to add plus-lighter blending during cross-fade. See issue 8924.
- Add mix-blend-mode to list of properties copied over to the '::view-transition-group'. See issue 8962.
- Add text-orientation to list of properties copied over to the '::view-transition-group'. See issue 8230.
- Refactor the old capture algorithm to properly set captured in a view transition before reading the value.

### § Changes from 2022-05-25 Working Draft

- Fix typo in ::view-transition-new user agent style sheet. See PR.

### § Changes from 2022-11-24 Working Draft

- Pointer events resolve to the documentElement when rendering is suppressed. See issue 7797.
- Add rendering constraints to elements participating in a transition. See issue 8139 and issue 7882.
- Remove html specifics from UA stylesheet to support ViewTransitions on SVG Documents.
- Rename updateDOMCallback to `UpdateCallback`. See issue 8144.
- Rename snapshot viewport to snapshot containing block.
- Skip the transition if viewport size changes. See issue 8045.
- Add support for :only-child. See issue 8057.
- Add concept of a tree of pseudo-elements under pseudo-element root. See issue 8113.
- When skipping a transition, the `UpdateCallback` is called in own task rather than synchronously. See issue 7904
- When capturing images, at least the in-viewport part of the image should be captured, downscale if needed. See issue 8561.
- Applying the ink overflow to the captured image is implementation defined, and doesn't affect the image's natural size. See issue 8597.
- Fragmented elements don't participate in view transitions. See issue 8339.
- Rename "snapshot root" to "snapshot containing block", and make it an absolute positioning containing block and a fixed positioning containing block for its descendants. See issue 8505.

### § Changes from 2022-10-25 Working Draft (FPWD)

- Add dynamic view transition style sheet concept for dynamically generated UA styles scoped to the current Document.
- Add snapshot viewport concept. See issue 7859.
- Clarify timing for resolving/rejecting promises when skipping the transition. See issue 7956.
- Elements under a content-visibility:auto element that skips its contents are ignored. See issue 7874.
- UA styles on the pseudo-DOM stay in sync with author DOM for any developer observable API. See issue 7812.
- Suppress rendering during updateCallback. See issue 7784.
- Changes in size/position of elements in the new Document generate new UA animation keyframes. See issue 7813.
- Scope keyframes to user agent stylesheets using -ua- prefix. See issue 7560.
- Update pseudo element names to view-transition*. See issue 7960.
- Update selector syntax for pseudo-elements. See issue 7788.
- Add sections for security/privacy considerations.

## § 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 7 ¶
>
> 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.

## § CR exit criteria

For this specification to be advanced to Proposed Recommendation, there must be at least two independent, interoperable implementations of each feature. Each feature may be implemented by a different set of products, there is no requirement that all features be implemented by a single product. For the purposes of this criterion, we define the following terms:

**independent**
: each implementation must be developed by a different party and cannot share, reuse, or derive from code used by another qualifying implementation. Sections of code that have no bearing on the implementation of this specification are exempt from this requirement.

**interoperable**
: passing the respective test case(s) in the official CSS test suite, or, if the implementation is not a Web browser, an equivalent test. Every relevant test in the test suite should have an equivalent test created if such a user agent (UA) is to be used to claim interoperability. In addition if such a UA is to be used to claim interoperability, then there must one or more additional UAs which can also pass those equivalent tests in the same way for the purpose of interoperability. The equivalent tests must be made publicly available for the purposes of peer review.

**implementation**
: a user agent which:

  1. implements the specification.
  2. is available to the general public. The implementation may be a shipping product or other publicly available version (i.e., beta version, preview release, or "nightly build"). Non-shipping product releases must have implemented the feature(s) for a period of at least one month in order to demonstrate stability.
  3. is not experimental (i.e., a version specifically designed to pass the test suite and is not intended for normal usage going forward).

The specification will remain Candidate Recommendation for at least six months.

## § Index

### § Terms defined by this specification

### § Terms defined by reference

- [] defines the following terms:
  - NavigateEvent
  - signal
- [COMPOSITING-1] defines the following terms:
  - isolation
  - mix-blend-mode
- [CSS-2022] defines the following terms:
  - style sheet
- [CSS-ANIMATIONS-1] defines the following terms:
  - CSSKeyframesRule
- [CSS-BACKGROUNDS-3] defines the following terms:
  - canvas background
- [CSS-BOX-4] defines the following terms:
  - border box
  - content box
- [CSS-BREAK-4] defines the following terms:
  - box fragment
  - fragment
- [CSS-CASCADE-5] defines the following terms:
  - computed value
  - user-agent origin
- [CSS-COLOR-4] defines the following terms:
  - opacity
- [CSS-CONTAIN-2] defines the following terms:
  - skips its contents
- [CSS-DISPLAY-3] defines the following terms:
  - containing block
  - initial containing block
  - principal box
  - replaced element
  - root element
- [CSS-DISPLAY-4] defines the following terms:
  - invisible
- [CSS-IMAGES-3] defines the following terms:
  - natural dimension
  - natural size
- [CSS-IMAGES-4] defines the following terms:
  - element-not-rendered
- [CSS-MASKING-1] defines the following terms:
  - clip-path
- [CSS-OVERFLOW-3] defines the following terms:
  - ink overflow
  - ink overflow area
  - ink overflow rectangle
  - overflow
- [CSS-POSITION-3] defines the following terms:
  - absolute positioning containing block
  - fixed positioning containing block
- [CSS-POSITION-4] defines the following terms:
  - top layer
- [CSS-PSEUDO-4] defines the following terms:
  - tree-abiding pseudo-element
- [CSS-SCOPING-1] defines the following terms:
  - flat tree
- [CSS-SIZING-3] defines the following terms:
  - height
  - width
- [CSS-TRANSFORMS-1] defines the following terms:
  - identity transform function
  - transform
- [CSS-TRANSFORMS-2] defines the following terms:
  - <transform-function>
- [CSS-VALUES-4] defines the following terms:
  - <custom-ident>
  - |
- [CSS-VIEWPORT] defines the following terms:
  - interactive-widget
- [CSS-WRITING-MODES-3] defines the following terms:
  - direction
- [CSS-WRITING-MODES-4] defines the following terms:
  - text-orientation
  - writing-mode
- [CSS2] defines the following terms:
  - pseudo-elements
  - stacking context

[CSS22] defines the following terms:
- element

[CSSOM-1] defines the following terms:
- CSSStyleRule

[CSSOM-VIEW-1] defines the following terms:
- layout viewport
- scrolling box

[DOM] defines the following terms:
- Document
- Element
- child
- connected
- descendant
- document
- document element
- inclusive descendant
- node document
- parent
- participate
- root
- shadow-including descendant
- sibling
- tree

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

[FILTER-EFFECTS-2] defines the following terms:
- backdrop root

[GEOMETRY-1] defines the following terms:
- height
- width

[HTML] defines the following terms:
- associated document
- dom manipulation task source
- queue a global task
- relevant global object
- relevant realm
- unhandledrejection

[INFRA] defines the following terms:
- append (for list)
- append (for set)
- assert
- contain
- continue
- exist
- for each (for list)
- for each (for map)
- implementation-defined
- list
- map
- set
- struct
- tuple
- values

[INTERSECTION-OBSERVER] defines the following terms:
- IntersectionObserver

[POINTEREVENTS3] defines the following terms:
- pointer capture

[SELECTORS-3] defines the following terms:
- *

[SELECTORS-4] defines the following terms:
- :only-child
- functional pseudo-element
- originating element
- originating pseudo-element
- pseudo-element
- selector
- type selector
- ultimate originating element

[WEB-ANIMATIONS-1] defines the following terms:
- associated effect
- document timeline
- effect target
- paused
- pending animation event queue
- play state
- running
- timeline

[WEBIDL] defines the following terms:
- AbortError
- DOMException
- Exposed
- InvalidStateError
- Promise
- TimeoutError
- a new promise
- a promise resolved with
- any
- getter steps
- invoke
- mark as handled
- method steps
- react
- reacting
- reject
- resolve
- this
- undefined
- unrestricted double

## § References

### § Normative References

**[COMPOSITING-1]**
Rik Cabanier; Nikos Andronikos. *Compositing and Blending Level 1*. 13 January 2015. CR. URL: https://www.w3.org/TR/compositing-1/

**[CSS-2022]**
Tab Atkins Jr.; Elika Etemad; Florian Rivoal. *CSS Snapshot 2022*. 22 November 2022. NOTE. URL: https://www.w3.org/TR/css-2022/

**[CSS-ANIMATIONS-1]**
David Baron; et al. *CSS Animations Level 1*. 2 March 2023. WD. URL: https://www.w3.org/TR/css-animations-1/

**[CSS-BACKGROUNDS-3]**
Bert Bos; Elika Etemad; Brad Kemper. *CSS Backgrounds and Borders Module Level 3*. 14 February 2023. CR. URL: https://www.w3.org/TR/css-backgrounds-3/

**[CSS-BOX-4]**
Elika Etemad. *CSS Box Model Module Level 4*. 3 November 2022. WD. URL: https://www.w3.org/TR/css-box-4/

**[CSS-BREAK-4]**
Rossen Atanassov; Elika Etemad. *CSS Fragmentation Module Level 4*. 18 December 2018. WD. URL: https://www.w3.org/TR/css-break-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]**
Tab Atkins Jr.; Chris Lilley; Lea Verou. *CSS Color Module Level 4*. 1 November 2022. 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-DISPLAY-3]**
Elika Etemad; Tab Atkins Jr.. *CSS Display Module Level 3*. 30 March 2023. CR. URL: https://www.w3.org/TR/css-display-3/

**[CSS-DISPLAY-4]**
*CSS Display Module Level 4*. Editor's Draft. URL: https://drafts.csswg.org/css-display-4/

[CSS-IMAGES-3]
Tab Atkins Jr.; Elika Etemad; Lea Verou. *CSS Images Module Level 3*. 17 December 2020. CR. URL: https://www.w3.org/TR/css-images-3/

[CSS-IMAGES-4]
Tab Atkins Jr.; Elika Etemad; Lea Verou. *CSS Images Module Level 4*. 17 February 2023. WD. URL: https://www.w3.org/TR/css-images-4/

[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-POSITION-3]
Elika Etemad; Tab Atkins Jr.. *CSS Positioned Layout Module Level 3*. 3 April 2023. WD. URL: https://www.w3.org/TR/css-position-3/

[CSS-POSITION-4]
*CSS Positioned Layout Module Level 4*. Editor's Draft. URL: https://drafts.csswg.org/css-position-4/

[CSS-PSEUDO-4]
Daniel Glazman; Elika Etemad; Alan Stearns. *CSS Pseudo-Elements Module Level 4*. 30 December 2022. WD. URL: https://www.w3.org/TR/css-pseudo-4/

[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-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-VALUES-4]
Tab Atkins Jr.; Elika Etemad. *CSS Values and Units Module Level 4*. 6 April 2023. WD. URL: https://www.w3.org/TR/css-values-4/

[CSS-VIEWPORT]
*CSS Viewport Module Level 1*. Editor's Draft. URL: https://drafts.csswg.org/css-viewport/

[CSS-WRITING-MODES-3]
Elika Etemad; Koji Ishii. *CSS Writing Modes Level 3*. 10 December 2019. REC. URL: https://www.w3.org/TR/css-writing-modes-3/

[CSS-WRITING-MODES-4]
Elika Etemad; Koji Ishii. *CSS Writing Modes Level 4*. 30 July 2019. CR. URL: https://www.w3.org/TR/css-writing-modes-4/

[CSS2]
Bert Bos; et al. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. 7 June 2011. REC. URL: https://www.w3.org/TR/CSS21/

[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/

[CSSOM-VIEW-1]
Simon Pieters. *CSSOM View Module*. 17 March 2016. WD. URL: https://www.w3.org/TR/cssom-view-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/

[FILTER-EFFECTS-2]
*Filter Effects Module Level 2*. Editor's Draft. URL: https://drafts.fxtf.org/filter-effects-2/

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

[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/

[INTERSECTION-OBSERVER]
Stefan Zager; Emilio Cobos Álvarez. *Intersection Observer*. 6 July 2022. WD. URL: https://www.w3.org/TR/intersection-observer/

[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/

[WEB-ANIMATIONS-1]
Brian Birtles; et al. *Web Animations*. 5 June 2023. WD. URL: https://www.w3.org/TR/web-animations-1/

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

## § Informative References

[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/

[POINTEREVENTS3]
Patrick Lauke; Navid Zolghadr. *Pointer Events*. 2 March 2023. WD. URL: https://www.w3.org/TR/pointerevents3/

## § Property Index

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

## § IDL Index

```
partial interface Document {
  ViewTransition startViewTransition(optional UpdateCallback? updateCallback = null);
};

callback UpdateCallback = Promise<any> ();

[Exposed=Window]
interface ViewTransition {
  readonly attribute Promise<undefined> updateCallbackDone;
  readonly attribute Promise<undefined> ready;
  readonly attribute Promise<undefined> finished;
  undefined skipTransition();
};
```