Captured Surface Switching - Working Doc

Abstract

This document is a working document to develop an API for captured surface switching and complements the discussion on the github issue thread: https://github.com/w3c/mediacapture-screen-share-extensions/issues/4

The intention is to provide a place to list terms, proposals and topics of discussions in a more structured way.

Status of This Document

This document is not complete. It is subject to major changes and, while early experimentations are encouraged, it is therefore not intended for implementation.

This document was published by the Web Real-Time Communications Working Group as a working document for collaboration on designing the API for captured surface switching.

As a working document, it does not imply endorsement by W3C and its Members.

This document 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.

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

Scope

In the first step, we would like to solve the following:

1. Cross-type surface-switching
2. Add audio-sharing after the fact
3. Frame delivery is clearly separated for the captured surfaces before and after the switch.

4. Cross-class surface-switching. Solve the polymorphism either by getting rid of it (specifically: BrowserCaptureMediaStreamTrack), or by surfacing tracks with the full capabilities of the current source

Definition of Terms

Surface Track

A surface track is a MediaStreamTrack whose source is a single captured surface.

Session Track

A session track is a MediaStreamTrack that spans a capture session from the time it is created, capturing media from any surface selected by the user during the session from that point on.

Hybrid Track

A hybrid track is a MediaStreamTrack that spans a capture session from the time it is created, capturing media from any surface selected by the user during the session from that point on, while maintaining feature parity with a surface track of the first source. Similar to a session track except not a feature-limited abstraction. Also similar to a surface track for the source at time of creation.. An alternative to the session/surface split.

Basic Models

Injection model

In the injection model, existing tracks are automatically updated to share the new captured surface selected by the user. Applications do not need to do anything for this to work.

This is what is used in the existing tab-switching functionality in Chrome and the window/screen-switching introduced in MacOS 14.

Switch-track model

To use the switch-track model, an application need to specify an opt-in, and add an event-listener:

```
getDisplayMedia({mode: "switch-track", …})
controller.onsourceswitch = event => {
  video.srcObject = event.stream;
};
```

When the user switches to a new surface:

1. Tracks capturing the old surface are ended.
2. An event is posted to the application with a new MediaStream containing MediaStreamTracks for the new surface.

Since old tracks end and new tracks are created for each switched surface, these tracks are by definition surface tracks.

Extensions to the Injection Model

The existing injection model as used for, e.g., tab-switching in Chrome has limitations when it comes to cross-type-switching, late addition of audio and keeping frames clearly separated between different captured surfaces.

This section presents a series of extensions to the model to mitigate these limitations.

Surface-switching events

Some applications need to react to surface switches. To that end, applications can listen to surface-switching events on the CaptureHandler:

```
controller.onsurfaceswitch = event => {
  // React to the surface switch
};
```

This event is fired after the properties have been updated to reflect the new captured surface for all tracks associated with the capture.

Cross-type surface switching

Some applications adapt their behavior to what type of surface they share. In the case of existing applications they typically check the type of the track returned from getDisplayMedia and then expect the type to remain constant throughout the capture. For backward compatibility with such applications, an opt-in should be specified to enable cross-type surface switching:

```
getDisplayMedia({allowCrossTypeSurfaceSwitching: “include”});
```

Initially, the default value for this option will need to be “exclude”, but hopefully, the default value can be changed to “include” at some point in the future.

Auto pause

Auto-pause is an extension to the injection model intended to avoid transmitting frames during the surface-switching transition period.

With auto-pause, frame-delivery is stopped before the surface switching occurs. After the surface-switch, applications are allowed time to set up the capture before frame-delivery is resumed.

Shape #1: Opt-in, events and enabled-false

This proposal consists of three parts.

1. Extend getDisplayMedia()'s input to allow opt-in

```
JavaScript
// ...getDisplayMedia({..., autoPause: true, ...});
partial dictionary DisplayMediaStreamOptions {
  boolean autoPause = false;
};
```

2. Define events that are fired when a switch happens

These events fire independently of `DisplayMediaStreamOptions.autoPause`'s value.

```
JavaScript
// controller.addEventHandler('surfaceswitch', ...);
partial interface CaptureController {
  EventHandler onsurfaceswitch;
};
```

3. Specify that auto-paused tracks get their enabled-field set to false.

Whenever the user interacts with the operating system and/or the user agent in a manner indicating their intention to change the captured surface, the user agent *MUST* perform the following steps:

1. Stop emitting frames on the old tracks.

2. Queue a task to:
    1. If autoPause is true, then for each track in CaptureController.[[AssociatedTracks]], set track.enabled to false. (This newly introduced internal slot, AssociatedTracks, contains all tracks returned by gDM and their clones.)
    2. Change the sources for the video track and potentially the audio track.
    3. Fire an `Event` on named `"surfaceswitch"` at this.

Shape #2

A possible API shape for this is that applications can add a handler CaptureController:

```
controller.postSurfaceSwitchingHandler = async () => {
  // Perform setup, then resolve the returned promise.
};
```

When the promise returned from the handler is resolved, frame-delivery is resumed.

If no handler has been set on the CaptureController, frame-delivery is resumed immediately.

Late audio

With the late-audio extension, a user agent is allowed to add audio to an ongoing capture after the capture has started. To do this, the user agent includes a muted audio track if the application requests audio and the user does not share audio, and if the user at a later time enables audio-sharing, the audio track is unmuted.

This behavior is already allowed according to the screen-capture spec [link]:

> If the user agent knows no audio will be shared for the lifetime of the stream it *MUST NOT* include an audio track in the resulting stream.

This means: If a user agent provides an option to enable audio sharing after the capture has started, it does not know that no audio will be shared for the lifetime of the stream, and it then MAY include an audio track in the resulting stream.

However, for backward compatibility with applications that use the presence of an audio-track to determine if a user shared audio or not, a hint should be added to enable this behavior:

```
getDisplayMedia({allowLateAudio: “include”});
```

Initially, the default value for this option should be “exclude”, but hopefully, the default value can be changed to “include” at some point in the future.

Hybrid Models

Multi-track model

The multi-track model combines the injection model and a modified switch-track model. The goal is to make both modes of operations available in parallel, while keeping things simple for applications that only need to use one of them.

With this model, getDisplayMedia returns a session track that works similarly to the injection model:

```
video.srcObject = await getDisplayMedia({...}); // session track
```

Applications that want to use the add-track model, can subscribe to an `onnewsource` event:

```
controller.onnewsource = event => {
  video.srcObject = event.stream; // surface tracks
};
await getDisplayMedia({controller, ...});
```

It is also possible to receive both these tracks in parallel:

```
controller.onnewsource = event => {
  video1.srcObject = event.stream; // surface tracks
};
video2.srcObject = await getDisplayMedia({controller, ...}); // session track
```

Note that, unlike `onsourceswitch` events, the `onnewsource` event is sent for every captured surface, including the first one. The reason for this is that `getDisplayMedia` needs to return the session-track used for the injection-model-like mode of operation.

The relationship between the session track and the surface tracks is discussed in the variants below

Multi-track model with a mixing-track

In the mixing-track variant, the session-track is a proxy-track for the underlying surface-tracks and just relays the media from them. Operations performed on a session track are delegated to the underlying surface track and changes to the surface tracks are exposed through the session track.

When the user switches to a new captured surface, the session track is switched over from the old tracks to the tracks for the new surface.

Polyfill sketch

The mixing-track can be polyfilled on top of the add-track model, at least for video tracks (using VideoTrackGenerator).

Sketch for illustrative purposes using nonstandard language:

```
//TODO: Replace MediaStreamTrackGenerator with VideoTrackGenerator
sessionTrack = new MediaStreamTrackGenerator("video");
controller.onnewsource = event => {
  processor = new MediaStreamTrackProcessor(e.stream.getVideoTracks()[0]);
  //TODO: release locks before second call to pipeTo
  processor.readable.pipeTo(sessionTrack.writable);
};
await getDisplayMedia({controller, ...});
```

Multi-track model with independent tracks

With independent tracks, the session track and surface tracks act as clones of each other.

To avoid burdening applications with stopping tracks that they don't need, the track types requested need to be specified in a parameter, e.g.,

```
controller.onnewsource = ({stream}) => {
  video.srcObject = stream;
};
await getDisplayMedia({controller, trackTypes: [“surface”], ...});
```

The trackTypes parameter can also work as an opt-in so it is still sufficient with one parameter to enable this model.

Late decision model

The late-decision model leaves the decision of injection up to the application at the time of the switch. The goal is to remain flexible to downstream needs and allow for an optimal decision based on the most recent information. No opt-in is required.

With this model, getDisplayMedia returns a hybrid track that supports the injection model (no event is fired at this time):

```
video.srcObject = await getDisplayMedia({controller, ...}); // contains
hybrid tracks
```

The application can register for events to learn of the user switching source (without committing to intervening):

```
controller.onsourceswitch = event => {
  if (!toInjectOrNot(video.srcObject, event.stream)) {
    video.srcObject.getTracks().forEach(track => track.stop());
    video1.srcObject = event.stream.clone(); // contains new hybrid tracks
  }
};
```

The UA stops the event tracks synchronously behind the event handlers who need to clone them if they wish to use them, for a clean handoff. The UA also holds back new content frames from injection tracks until that point.

The app can make a different decision on whether to inject or not each time the user switches source.

The user agent is responsible for updating the properties of the track to match its new source, including making sure any no-longer-supported methods (e.g. cropTo()) fail with InvalidStateError.

Topics for discussion

This section is intended for topics that apply to multiple different models so that they can be discussed in one place.

Events vs Callbacks

TBD

Preventing content bleed during switch

TBD

Use Cases

1. Presenting the Capture in a Video Element

This example is intended to capture the simple case of a capture-agnostic application that just wants to present the video part of the capture.

1a. Injection Model

```
video.srcObject = await getDisplayMedia();
```

1b. Switch-track Model

```
getDisplayMedia({controller, mode: "switch-track", …})
controller.onsourceswitch = event => {
  video.srcObject = event.stream; // flickers from HTMLVideoElement’s load
algorithm
};
```

1c. Multi-track model with a mixing-track

```
video.srcObject = await getDisplayMedia({controller, mode: “multi-track”,
...});
```

1d. Multi-track model with independent tracks

```
video.srcObject = await getDisplayMedia({controller, trackTypes: “session”,
...});
```

1e. Late decision model

```
video.srcObject = await getDisplayMedia();
```

2. Write each surface into a separate file

2a. Injection model with Events and Auto-pause

```
stream = await getDisplayMedia({controller, audio: true, …});
mediaRecorder = new MediaRecorder(stream);
controller.onsourceswitch = event => {
  mediaRecorder.stop()
  mediaRecorder = new MediaRecorder(stream);
}
```

Each recording will start after a new surface has been selected, and auto-pause guarantees that no old frames will be written.

2b. Switch-track Model

```
getDisplayMedia({controller, audio: true, mode: "switch-track", …});
controller.onsourceswitch = event => {
  new MediaRecorder(event.stream);
}
```

2e. Late decision Model

```
let stream = await getDisplayMedia({controller, audio: true});
controller.onsourceswitch = event => {
  stream.getTracks().forEach(track => track.stop());
  new MediaRecorder(stream = event.stream.clone());
}
controller.onsourceswitch({stream});
```

3. Write an entire session into a single file

3a. Injection Model

```
new MediaRecorder(await getDisplayMedia({controller, audio: true});
```

3e. Late decision Model

```
new MediaRecorder(await getDisplayMedia({controller, audio: true});
```

4. VC: Transmit if predicate(surface), pause while !predicate(surface)

4a. Injection model with Events and Auto-pause

```
controller.onsourceswitch = event => {
  peerConnection.getSenders()[0].track.enabled =
predicate(event.stream);
}
```

4b. Switch-track Model

Comment moved

```
controller.onsourceswitch = async event => {
  // For simplicity, assume video-only.
  const [videoTrack] = event.stream.getVideoTracks();
  videoTrack.enabled = predicate(event.stream);
  await peerConnection.getSenders()[0].replaceTrack(videoTrack);
}
```

4e. Late decision model

```
controller.onsourceswitch = event => {
  peerConnection.getSenders()[0].track.enabled =
predicate(event.stream);
}
```

5. VC: Share X1 then X2 over a peer connection, potentially adding or removing audio

Assume a pre-negotiated audio sender.

5a. Injection model with Late audio

```
peerConnection.call(..., await getDisplayMedia(...));
```

With Late-audio, any added audio will be injected into the existing audio-track and the audio-track becoming unmuted. If audio is removed, the audio-track will be muted.

5b. Switch-track Model

```
controller.onsourceswitch = async event => {
  const [videoTrack] = event.stream.getVideoTracks();
  const audioTrack =
      event.stream.getAudioTracks().length > 0 ?
      event.stream.getAudioTracks()[0] :
      null;
  await Promise.all([
    peerConnection.getSenders()[0].replaceTrack(videoTrack),
    peerConnection.getSenders()[1].replaceTrack(audioTrack)
  ]);
}
```

5e. Late decision model

```
controller.onsourceswitch = async event => {
  peerConnection.getSenders().forEach(({track}) => track.stop());
  const [videoTrack] = event.stream.getVideoTracks();
  const [audioTrack] = event.stream.getAudioTracks();
  await Promise.all([
    peerConnection.getSenders()[0].replaceTrack(videoTrack),
    peerConnection.getSenders()[1].replaceTrack(audioTrack)
  ]);
}
```

6. Switching between tabs where one is cropped (Region Capture)

A web page that crops to a content-area when self-capturing, but leaves other surfaces uncropped.

6a. Injection Model with Auto-pause

```
stream = await getDisplayMedia({controller, …});
controller.postSurfaceSwitchingHandler = async () => {
  if (isSelfCapture()) {
    return stream.getVideoTracks()[0]
           .cropTo(contentAreaCropTarget);
  }
};
```

6b. Switch-track Model

```
controller.onsourceswitch = async event => {
  if (isSelfCapture()) {
    await event.stream.getVideoTracks()[0]
          .cropTo(contentAreaCropTarget);
  }
  sink = event.stream;
};
```

6e. Late decision Model

```
controller.onsourceswitch = async event => {
  if (isSelfCapture()) {
    await event.stream.getVideoTracks()[0]
```

```
          .cropTo(contentAreaCropTarget);
  }
  sink.getTracks().forEach(track => track.stop());
  sink = event.stream;
};
```

7. Playback of Audio and Video Capture in a Video Element

This is intended to capture switching from a source without audio to one with audio, for more rare local playback use cases like audio visualizers or audio modifier apps designed for headsets.

7a. Injection Model with Late Audio

```
video.srcObject = await getDisplayMedia({audio: true});
```

With Late-audio, any added audio will be injected into the existing audio-track and the audio-track becoming unmuted. If audio is removed, the audio-track will be muted.

7b. Switch-track Model

```
getDisplayMedia({audio: true, controller, mode: "switch-track", …})
controller.onsourceswitch = event => {
  video.srcObject = event.stream; // flickers from HTMLVideoElement’s load
algorithm
};
```

7e. Late decision model

```
video.srcObject = await getDisplayMedia({audio: true, controller});
controller.onsourceswitch = event => {
  // avoids flicker from HTMLVideoElement’s load algorithm
  const [oldtrack] = video.srcObject.getAudioTracks();
  const [newtrack] = event.stream.getAudioTracks();
  if (!oldtrack && newtrack) {
    video.srcObject.addTrack(newtrack.clone());
  }
}
```

8. Write similar parts of a session into a single file, but separate files for parts that differ on application-determined criteria, e.g. capture type, resolution, presence of audio, origin?

An application might want to keep recording into the same file or switch to a new file based on what the user switched to and/or from.

8a. Injection model with Events and Auto-pause`stream = await getDisplayMedia({controller, audio: true, …});`

```
mediaRecorder = new MediaRecorder(stream);
controller.onsourceswitch = event => {
  if (!toKeepRecordingOrNot(stream)) {
    mediaRecorder.stop()
    mediaRecorder = new MediaRecorder(stream);
  }
}
```

8e. Late decision Model

```
let stream = await getDisplayMedia({controller, audio: true});
new MediaRecorder(stream);
controller.onsourceswitch = event => {
  if (!toKeepRecordingOrNot(event.stream)) {
    stream.getTracks().forEach(track => track.stop());
    new MediaRecorder(stream = event.stream.clone());
  }
}
```

9. Diverging downstream needs from different consumers in the app: preview, transmitter, and recorder

A VC app might have multiple consumers of the same capture session through cloned tracks (combining the other use cases). E.g. a preview area (use case 1), a transmission component (use cases 4 and 5), and a record function (use case 3 or 7). The app might prefer a different choice for each component, specifically injection for the recorder when possible, and new tracks for everything else.

9c/d. Multi-track model

```
controller.onnewsource = event => {
  video.srcObject = event.stream;
  const [videoTrack] = event.stream.getVideoTracks();
  await peerConnection.getSenders()[0].replaceTrack(videoTrack);
};
new MediaRecorder(await getDisplayMedia({controller, /*opt-in*/});
```

where the opt-in is

- `mode: “multi-track”` for the mixing-track variant, and
- `trackTypes: [“session”, “surface”]` for the independent-tracks variant.

9e. Late decision Model

```
const stream = await getDisplayMedia({controller, audio: true});
const preview = new PreviewComponent(stream, controller);
const recorder = new RecorderComponent(stream.clone(), controller);
const transmit = new TransmissionComponent(stream.clone(),
controller);

// Each component can independently choose injection or not

class Component {
  constructor(stream, controller) {
    this.stream = stream;
    controller.addEventListener('sourceswitch',
this.handler.bind(this));
  }
}
class PreviewComponent extends Component {};
class RecorderComponent extends Component {};
class TransmitComponent extends Component {};
```