Chromium Stylus Writing into Input Design

Authors: flackr@chromium.org, peconn@chromium.org May 2022

Overview

Users should be able to write text over inputs on the page in order to write into the input. Sites must also be able to prevent this if they are handling stylus input.

Possible Stylus actions

There are several possible actions which should be supported for users using a stylus system. In some circumstances the same user action could be used to start multiple gestures. In order of precedence, a user moving the stylus should do the following:

- Custom developer event handling
- Writing into input fields
- Scroll the page

Developer event handling

Aside from specific security exception cases, developers should always be able to handle input events instead of any default system handling. For stylus, examples include an application that supports drawing, annotating on top of web content, or a site or library providing its own richer custom writing experience.

Example signature field which captures all drawn strokes when used by pen but allows text entry with other input devices.

Writing

Writing into input fields should be easy even in scrolling areas. For this reason, we should probably prefer writing into input fields over scrolling.

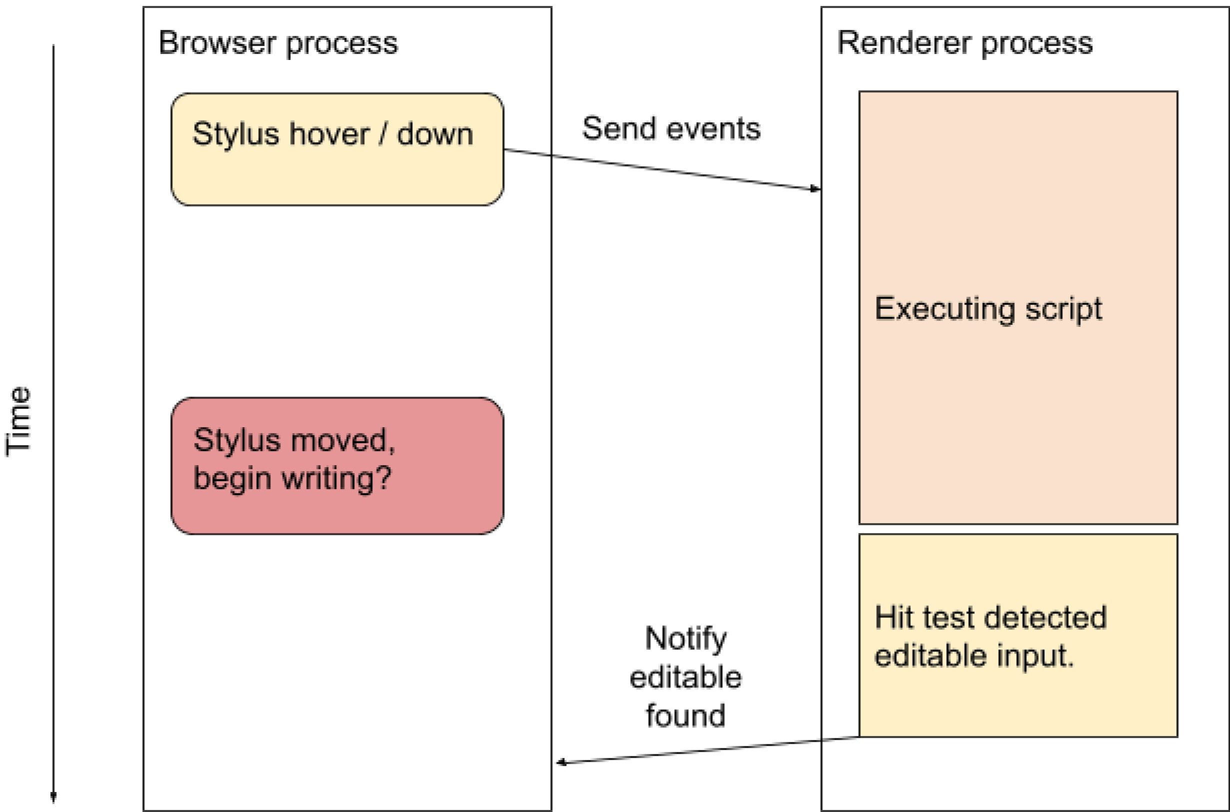
If the user had no other input mechanism by which to scroll the page, we would have to go the other way for large contenteditable pages (example page), as lacking any mechanism for scrolling the page could lead to users being unable to access the full page contents. It's possible in such circumstances we may want to require focus first before writing can take place in such cases.

Scrolling / text selection

If nothing else has prevented the stylus' events, it should perform the user agent's default stylus action whether that is scrolling or text selection.

Timing constraint

Any time information is needed from the renderer, it could take arbitrarily long to get this information. This means that the design should not attempt to make immediate decisions in the browser process based on input but rather must wait for information from blink, for example, consider the following timing:



In this case, when the stylus begins moving, the browser process does not yet know whether the stylus is over an editable field. The decision whether to begin writing must be delayed.

As an example, visit https://output.jsbin.com/vitibeg and try to write into the input field.

Design

The design is broken down into four main areas: beginning writing, continued events once writing starts, updating the editable field when text has been recognized, and finishing writing.

Beginning writing

InputRouterImpl

When the pen goes down on the screen, InputRouterImpl (browser process) calls FilterAndSendWebInputEvent to determine the disposition of the touch. This is responded to with a TouchEventAck giving the allowed touch actions (i.e. default browser actions) of the page at the touch location. Assuming that writing should be preferred to scrolling, we should disable all scrolling gestures from the allowed touch action when over an editable field.

While we are awaiting the allowed touch action, we should return `FilterGestureResult::kFilterGestureEventDelayed` from `TouchActionFilter::FilterGestureEvent` to accumulate all movement until the allowed action is known (i.e. when `OnSetTouchActionIsCalled`) at which point they are either dispatched as scrolling or as initial input to the stylus writing API or dropped if the developer is handling them.

Once we have received the known touch action ~~and processed the GestureScrollBegin (i.e. indicating sufficient movement to begin writing)~~ we can call out to the platform writing interface either right away or once we have also seen the GestureScrollBegin to begin handling the event stream including the events observed so far. See next section for discussion on this point.

Note: For stylus devices which want to show whether writing can commence on hover we may want to query the effective touch action on hover.

When to begin writing

We could begin writing at the point of GestureScrollBegin or once we know the touch action for the pointerdown event does not prevent "scrolling" and we have an input field.

Options (Number 2 is implemented):

1. In response to **touch action** from pointerdown. This will initiate writing even with regular taps, which means that subsequent stylus movement on screen until it is dismissed will be consumed by the writing service and translated into text for this input. This has the disadvantage that stylus taps lead to the writing service stealing subsequent taps which could be a bad user experience (e.g. they may want to

2. Once we have the **touch action** and the **GestureScrollBegin**. This means that if the user simply taps on an input field blink will continue to get the first opportunity to process subsequent events.

There are many cases on the web where tapping into an input field shows additional UI which the user may want to interact with, beginning writing right away makes it difficult to interact with this UI as tapping anywhere immediately after can be interpreted as wanting to write a period. For example, autofill suggestions typically show in a suggestion box after tapping into the input box. Sometimes developer UI has the same, for example the to input box on gmail, or this simple example showing the pattern

Blink Renderer

In addition to removing scrolling from the allowed touch action when we should be writing into an input field, blink should also set the element to be written into. This could either be done by (Option 2 is implemented):

1. Focusing the editable element to be written into (FocusedElementChanged)

2. Adding an API similar to focus which would allow InputRouterImpl / the writing service to later focus the element once writing had been confirmed. This method would also support hover.

Regardless of which method we use, the browser should be able to determine the bounds of the element using the WebElement::BoundsInViewport API.

Proximity support

The blink renderer may find nearby input fields so that the user doesn't have to start directly over them. This is similar to what is done with touch inputs where we find clickable elements within the touch size of the center point.

In particular, we would implement this in the PointerEventManager::AdjustTouchPointerEvent method (after renaming it to be Touch agnostic) by finding nearby nodes which may support input. We will have to be careful that this does not move the point away from a node which intended to handle it, e.g. a drawing application with a text field above should still be drawable to the top of the canvas next to the text field.

Continued writing

Once we begin writing, blink no longer needs to see the events. We can either:

- stop sending any events to blink and send all events directly to the writing service
- allow the writing service to receive events using a more direct path

Committing updates to blink

Regular text insertion at cursor

Once the writing service has recognized some text, it should use existing IME methods to send this text to blink, e.g.

- SetComposingText to hint at partially recognized text.
- CommitText once recognized text should be finalized.
- ExecuteEditCommands could support various richer text modification cases.

If required we could add new IME methods for as-yet-unsupported editing operations.

Writing gestures

Gestures translate an action over some physical region of the screen to edits on the text at that location. For example, the following would erase the word "so" from the sentence:

This is ~~so~~ neat

As such, they require knowledge of the text layout in order to determine the text affected by the gesture. There are three ways this could be implemented in chromium, of which I think the best choice is option 3, sending the positional gesture to blink:

NOTE: Option 3 is implemented in blink

Option 1: Synchronous handling in browser

In order to synchronously determine the text modification this would require that the browser know the specific location of the text. This would require blink sending all of the text locations for at least the actively edited element as a message through RenderFrameImpl (similar to SetSelectedText).

Advantages:

- Simple IME messages can be sent to the renderer
- Matches native input interfaces
- IME can differentiate between locations with text and no text

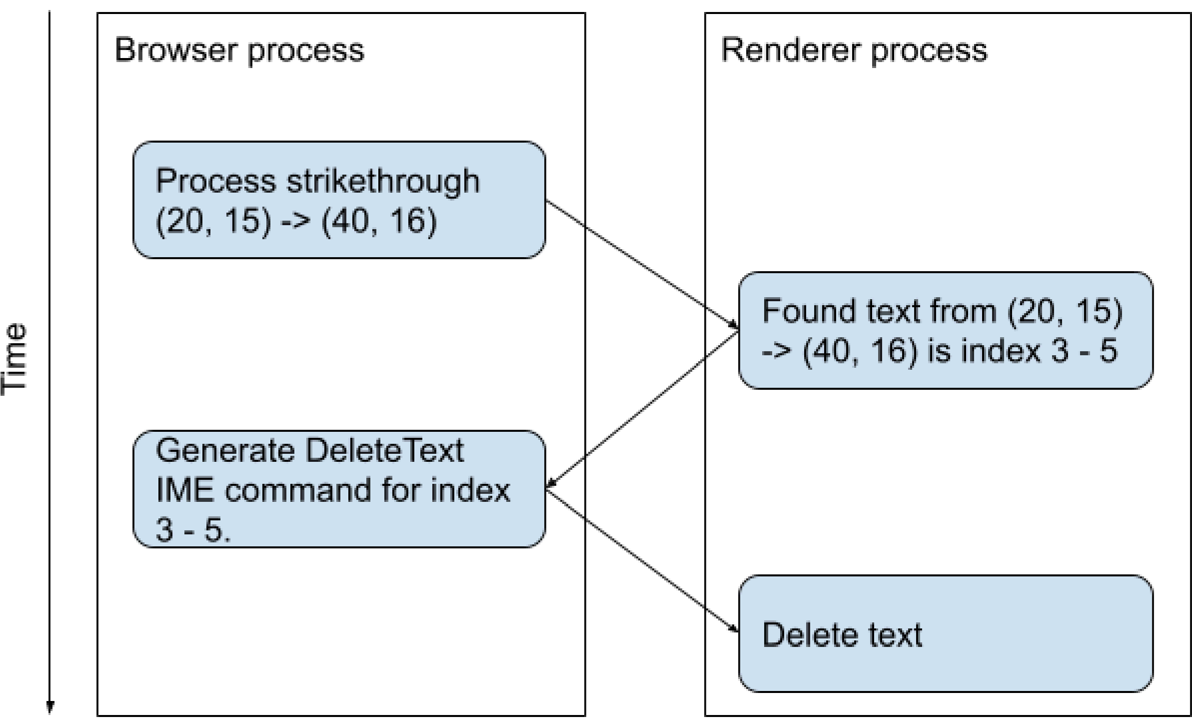
Disadvantages:

- Passing all of this extra layout information from blink to the browser will be expensive,
- The information necessary to represent the text accurately can include arbitrary transforms and clips and
- **writing can begin on a field that is not yet focused. It is not possible to ensure we have the information in time to handle the gesture as the renderer main thread can be arbitrarily delayed (see timing section).**

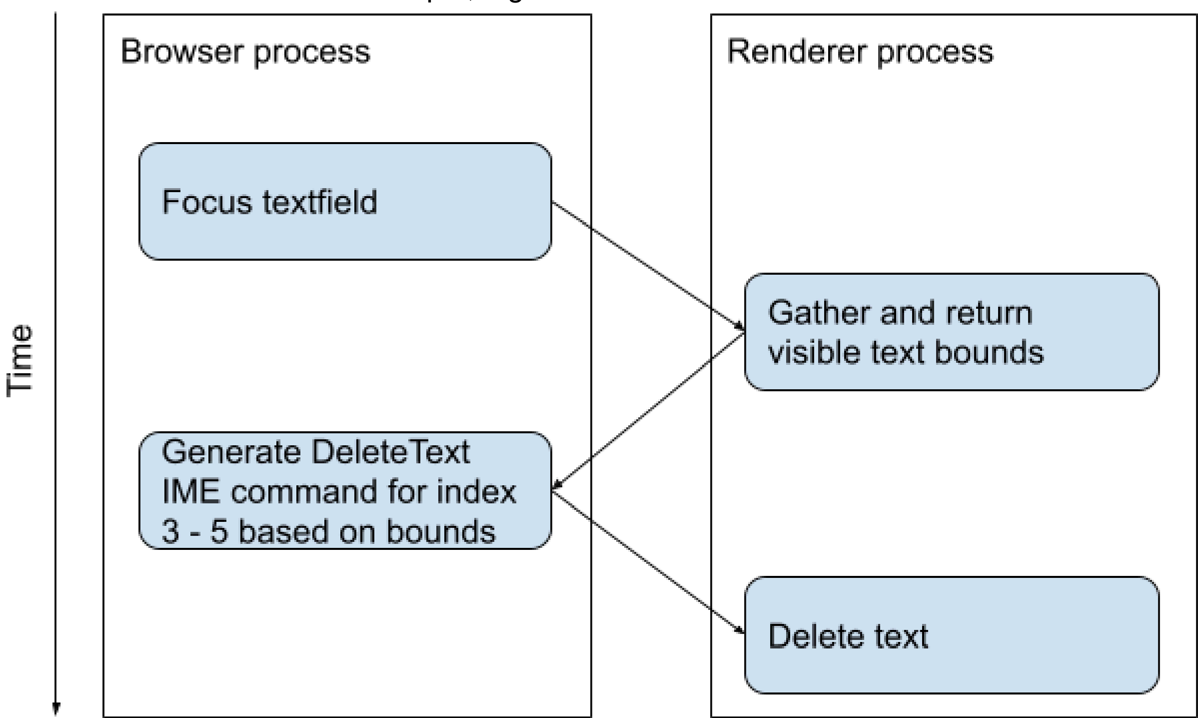
Due to the race to get the information from blink at the start of a gesture, we should not pursue this approach.

Option 2: Asynchronous handling in browser

Once a stylus gesture has begun, the browser could request information from blink about the text location of the gesture position, e.g.



There is an existing mechanism in the code when the IME calls requestCursorUpdates (codesearch) where blink will send the character bounds for the current composition (produced by the GetCompositionCharacterBounds method). We could augment this to also include character bounds for all on screen characters. This experimental patch tests sending **all** character bounds for the current input, e.g.



Advantages:

- Simple IME messages can be sent to the renderer
- IME can differentiate between locations with text and no text
- Supporting additional gestures doesn't require new gesture messages

Disadvantages:

- Extra round trip required (though could be triggered as soon as the pen goes down)
- Extra APIs required for finding text at location, negates the advantage of keeping the IME commands simple.

Given the disadvantages, we should not pursue this approach either.

Option 3: Sending gestures with coordinates to renderer

Sending IME messages with location based gestures means that blink can synchronously determine the affected text and perform the committed edits. We can create a ScreenGesture messages and send this to blink over the IME interface, each containing the minimal information required to execute the gesture. E.g. ScreenGesture could be a union of possible gestures such as:

- Strike out text { Point start, Point end }
- Remove space { Point point, direction? }
- Insert text { Point point, text, direction? }

A success / failure callback could be triggered from blink to inform the browser whether there was text at the target location such that something that could be interpreted as either a gesture or text could be reinterpreted as text if there was no text to be affected by the gesture, e.g.

Input	Handling
This is so neat	Browser sends Strike out { (30, 15), (45, 15) } Blink removes "so", responds with success.
This is so neat	Browser sends Strike out { (80, 15), (95, 15) } Blink responds with failure Browser recognizes it as text and inserts "-"
This is so neat	Browser sends Insert text { (40, 5), " ", BELOW } Blink inserts a space between "s" and "o", responds with success.
This is so neat	Browser sends Insert text { (100, 10), " ", BELOW } Blink responds with failure Browser recognizes it as text and inserts "v".

Or, an alternate action could be passed along to take if a gesture was not recognized. E.g. insert "v" if the insert space action fails because there was no text at the drawn location.

Advantages:

- Keeps layout information in blink.

- Easily extensible to future gestures.

Disadvantages:

- Complicates IME interface, though this seems unavoidable.

Summary of options

The above options would result in the following API to be called by the Stylus writing framework (this is a rough example for illustration only):

The deleteText and insertText methods already exist.

Option	API shape

1	`TextInfo getTextAt(Rect area)`
	
	`// deleteText or insertText is called.`

2	`void getTextAt(Rect area, Callback<TextInfo> onGotText)`
	
	`// deleteText or insertText is called after onGotText is run.`

3	`void strikeOut(int x1, int y1, int x2, int y2,`
	`               Callback<Void> onNoTextToStrikeOut)`
	
	`// insertText is called if onNoTextToStrikeOut is run.`

4	`boolean strikeOut(int x1, int y1, int x2, int y2)`
	
	`// insertText is called if false is returned.`

Finishing writing

The writing service is responsible for recognizing when writing should finish and sending a message to chrome. This may be:

- A timeout
- ~~Tapping without dragging~~ (used for entering full stop / period, i.e. “.”)
- Some other onscreen UI

This should update the state in InputRouterImpl so that we can again scroll content / deliver events to blink.

Interfacing with the Stylus framework

There are two different APIs that we need to work with - the Android handwriting API Android introduced in Android T (13), and the DirectWrite API used on some Samsung devices.

The differences between these APIs should be abstracted behind a common interface and only the high level Chromium Java code should need to know which is being used.

Android's API

The Android API consists primarily of the startStylusHandwriting method, which should be called once we've determined writing has begun. Once that method has been called, we receive an `android.view.MotionEvent#FLAG_CANCELED` event and further user input is sent directly to the IME.

Other methods of note are View#setAutoHandwritingEnabled which can be used to disable the default handwriting triggering, and CursorAnchorInfo.Builder#setEditorBoundsInfo which informs the IME where the text editing area is.

TODO: Find and document the default triggering behaviour.

Android's API does not support gestures at the moment.

DirectWrite API

The DirectWrite API (demonstrated here) uses a Service connection. When the user starts writing, the browser would connect to the DirectWriting Service and forward input events to it.

TODO: Who is responsible for ending the writing session?

The DirectWriting Service sends gestures through a callback, supporting:

- GESTURE_TYPE_BACKSPACE
- GESTURE_TYPE_V_SPACE
- GESTURE_TYPE_WEDGE_SPACE
- GESTURE_TYPE_U_TYPE_REMOVE_SPACE
- GESTURE_TYPE_ARCH_TYPE_REMOVE_SPACE

Combined API

Event	Android API action	DirectWrite API action
App launch	setAutoHandwritingEnabled(false)	Check if DW is enabled. Fetch the DW parameters (the touch thresholds).
User hovers with the stylus		Eagerly connect to DW Service.

ACTION_HOVER_ENTER		
User begins writing	startStylusHandwriting setEditorBoundsInfo	Ensure connected to DW Service. Save input events until connection is established. Forward touch events to Service.
User stops writing		
Window receives onStop		Disconnect from the Service?

Considerations

Privacy

There are three types of data that Chromium may be sending to the handwriting recognition service:

- The input events of the user's handwriting.
- The editable text near where the user is handwriting.
- The bounds of the editable text area on the screen.

In the middle case, the information about the editable text near the user's cursor is already sent to the IME through calls such as InputConnection#getSurroundingText. This is used for IME features such as autocomplete. For handwriting, this is used to support gestures - for example, recognising the 'v' user gesture and determining where in the text a space needs to be inserted.

Android's API

With the design of Android's API, Chrome calls startStylusHandwriting and Android forwards the input events to the IME.

Samsung's API

Samsung's API is available only on Samsung devices, so Samsung would be able to intercept and record/log the user input information before Chrome received it, if it wanted to. In addition, the DirectWrite service will only be used if Samsung's keyboard is enabled.

In addition, Samsung says that their DirectWriting service does not collect user information.

Security

Android's API

Once startStylusHandwriting has been called, the Android framework forwards motion events directly to the IME, effectively taking Chrome out of the process. Once the IME has recognised the text, it sends the results through standard InputConnection calls.

Samsung's API

We will check the signature of the DirectWriting service once connected to ensure that it is the genuine app, published by Samsung.