Technical Disclosure Commons

Defensive Publications Series

December 2020

Gesture Disambiguation in Operating Systems

Jaeheon Yi

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation
Yi, Jaeheon, "Gesture Disambiguation in Operating Systems", Technical Disclosure Commons, (December 10, 2020)
https://www.tdcommons.org/dpubs_series/3872

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.

Gesture Disambiguation in Operating Systems

ABSTRACT

A visual operating system features multiple simultaneous clients displayed over a system user interface (UI). Examples of simultaneous clients are the accessibility client, the system UI, browsers, map-applications, etc. A touch gesture made by the user can be intended to be directed towards any one of the clients. However, in an architecture where no single component or client has global knowledge or ownership of all gestures, the client towards which the gesture should be directed is sometimes unclear. This disclosure describes techniques to disambiguate user gestures which can be incorporated in an operating system. The techniques enable simultaneously-running clients to define and claim gestures; provide a centralized mechanism to deny gestures; provide an expression of priority amongst competing clients; provide a mechanism for a parent client to defer a gesture to a child client; etc. Simultaneous access of a gesture by competing clients eliminates latency across the full breadth of client priorities.

KEYWORDS

- Gesture disambiguation
- Gesture recognition
- Multi-finger gesture
- Touch input
- Touch event
- Operating system

BACKGROUND

A modern operating system, e.g., one that can drive a touch display, features a multi-component, multi-runtime environment, e.g., multiple simultaneous applications, or clients, displayed over a system user interface (UI). Examples of simultaneous clients are the accessibility client (e.g., touch to magnify or touch to read aloud), the system UI (which manages other windows on the screen), browsers, digital map applications, social media applications, etc.

A touch gesture made by a user, e.g., a tap, a double-tap, a triple-tap, a left-swipe, a right-swipe, a two-finger pinch, an up-swipe, a down-swipe, a long-touch, etc., can be intended to be directed towards any one of the simultaneous clients. However, in an architecture where no single component or client has global knowledge or ownership of all gestures, the client towards which the gesture is to be directed is sometimes unclear. For example, a right-swipe made by a user viewing a digital map on a touchscreen can reasonably be interpreted as either a command to pan around the map to see its western regions, or a command to dismiss the map application. The problem of assigning gestures to clients is complicated by the small size of touchscreens relative to the human finger.

A scene manager creates a hierarchical *view tree*, and each UI client creates a view within that view tree. A view tree has a distinct *root view*. Each view owns a convex, rectilinear space, and these spaces form a spatial containment hierarchy down the view tree. A view space S thus serves two purposes: it enables the spatial manipulation of visual content in S (a positive freedom), and it guarantees that the ancestor views of S have the ability to manipulate S without interference (a negative freedom).

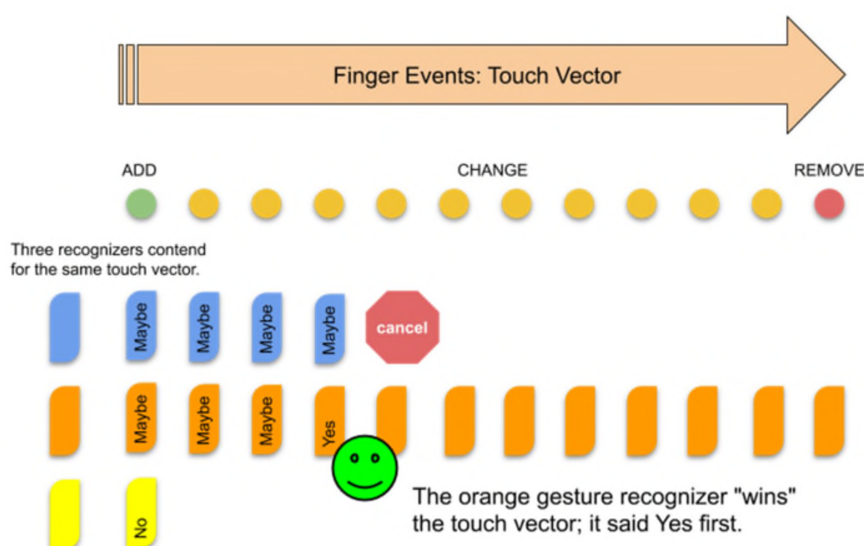

Fig. 1: A touch vector and recognizers that contend for it

Touch events are a tuple (t, f, p, x, y) where t is a timestamp, f identifies a finger (e.g., one of a multi-finger gesture), p identifies a phase, and (x,y) correspond to surface coordinates. Fig. 1 illustrates a vector of touch events issued by a single finger and sorted by timestamp. As illustrated in Fig. 1, a touch vector starts with the `ADD` event, which is when the user touches the screen to start a touch gesture. A complete touch vector ends with the `REMOVE` event, which is when the user removes the finger from the screen to end the touch gesture, or a `CANCEL` event (explained further below).

A *gesture recognizer*, illustrated in Fig. 1, is a module run by a client that contends for a user gesture. Mathematically, a gesture recognizer is a correspondence relation that relates an event in a set of touch vectors to a recognizer state. The set of states can be, for example, `{No, Maybe, Yes}`, `{No, Maybe, Yes, Hold}`, etc.

In Fig. 1, the blue recognizer accepts a long swipe for its client, and the yellow recognizer accepts a tap for its client, while the orange recognizer accepts a tap, a double-tap, a short-swipe, a long-swipe, etc. Given an ongoing touch vector, the yellow recognizer rapidly reaches the state `No`, e.g., declines to accept the touch vector as a legitimate input for its client. The blue recognizer starts with the state `Maybe`, indicating that the ongoing touch vector might possibly be a legitimate input for its client. The orange recognizer starts with a `Maybe` and changes to a `Yes`, indicating that the touch gesture is a legitimate input (short-swipe) for its client. The orange recognizer receives ownership of the gesture. Upon the orange recognizer issuing a `Yes`, the blue recognizer receives a `CANCEL` event and is withdrawn from contention.

A gesture recognizer is said to *consume* (accept as input) a touch vector and *outputs* a recognizer state for each event in the vector. A set of legitimate gestures for a given gesture recognizer is a set of touch vectors that end in `Yes`.

Given a set of gesture recognizers and a set of touch vectors, the *gesture disambiguation* problem is to partition a set of touch vectors amongst a set of recognizers. There is one winning recognizer for each touch vector. The partition may not happen atomically in time but typically happens before a touch vector completes. A *gesture arena* is where a set of recognizers consume a single touch vector to determine the winner on a per-finger basis. The arena winner receives the remainder of the touch vector; other recognizers receive a gesture cancellation event generated by the arena.

Recognizers may output `Yes`; the first such claimant with the fewest amount of `CHANGE` events is the winner. Recognizers may output a `No`; the arena drops these from further consideration. Recognizers may choose to output `Maybe`; in the absence of a recognizer claiming `Yes`, the arena will grant the win to the singleton-survivor `Maybe`. If multiple `Maybe`s persist until the `REMOVE` event (the sweep), the arena can choose one of these `Maybe`s as the winner. The `Hold` state is used by a recognizer to indicate an initial claim on a touch event, to be confirmed if the touch event evolves in a particular way. The `Hold` state can be used, e.g., by a recognizer that accepts a double-tap but has thus far seen only one of the expected two taps. If the second tap arrives within an expected time, the `Hold` converts to a `Yes`; else it converts to a `No`.

Conflicts, resolution, and priority

Two recognizers conflict if each outputs `Yes` on the same touch vector. For example, two right-swipe gestures conflict. Also, a tap and double-tap will naively conflict on the first tap. Conflicts make disambiguation difficult because the result (which recognizer wins) depends on implementation details such as timing (as in the former example) or aggressiveness in claiming a win (as in the latter). The emergent behavior can be nondeterministic and fragile as each

recognizer implementation evolves, which is undesirable. Hence there is a need for a way to resolve such conflicts more deterministically.

Conflicts can possibly be resolved *cooperatively* by making one of the gestures more passive or aggressive than the other. For example, defining a tap to be passive (outputs only `Maybe`) means the tap will win only if the double-tap outputs `No`. Another example is getting the gesture (e.g., right swipe) of one (parent) view to take precedence over the same gesture for another (child) component; the parent view can be tuned to be slightly more aggressive in claiming a win (although this may not work in practice). This does not require anything extra from the arena; it is a cooperative, a priori resolution scheme.

However, cooperation can be difficult to manage, and all actors may not be cooperative. Non-cooperative actors can be dealt with by encoding a notion of priority on the recognizers tied to the view hierarchy, e.g., ancestor views have higher priority, descendant views have lower priority, etc. For example, accessibility taps in a screen corner can take precedence over everything else; the system UI can be of a mid-level priority; and applications (e.g., browsers, map-applications, etc.) can be of low priority. A child of an application, e.g., a digital map spawned by a browser, can be of even lower priority. This can be achieved by having the accessibility manager output a view just below the root view, and the arena can ignore lower-priority claims until all recognizers from ancestor views output `No`. However, high-priority recognizers can induce latency for lower-priority recognizers. A developer of a high-priority recognizer can take special care to not accidentally introduce sources of gesture latency (e.g., a triple-tap), but such developer dependence is not robust. Also, *all* recognizer implementations are required to voluntarily *not* act on their claim of winning until the arena confirms a claim — a source of latency that may be unacceptable.

Gesture Types

There are two broad classes of gestures: *direct manipulation gestures* and *spatial gestures.* Direct manipulation gestures act on visible objects on the screen; e.g., a swipe to dismiss a surface, moving a slider back and forth, selecting a text region, etc. Spatial gestures are not tied to a particular object; e.g., a swipe to move to the next screen.

In order to effect direct manipulation of a view's object, the touch event must be interpretable in the view's coordinate system. In contrast, a spatial gesture can be encoded with more abstract coordinates.

The accessibility feature of magnification requires gestures to be invariant under scale change, e.g., a swipe should look like a swipe no matter the magnification factor. Otherwise, gesture recognizers need to take the magnification factor into account when attempting to recognize a swipe. Scale-invariant gestures are enabled by delivering touch events in an abstract space and by supplying a view-specific transform that maps the gesture to the view's coordinate system.

DESCRIPTION

This disclosure describes operating system techniques to disambiguate user gestures, e.g., determine accurately the client that the user intended their gesture for. Mechanisms are provided to enable privileged actors, e.g., accessibility programs and system UIs, to create a cohesive and performant gesture experience, while also enabling individual clients to define their own unique gestures, even, say, an A-shaped gesture. The techniques are flexible and suited to design practices that can change over time. The techniques enable various simultaneously-running clients to claim gestures; a centralized mechanism to deny gestures; an expression of priority amongst competing clients; a mechanism for a parent client to defer a gesture to a child client;

etc. The competing clients assess a gesture simultaneously, such that parent clients do not induce a latency on gestures intended for child clients.

Per the techniques, a scene-manager owns a global arena to disambiguate each touch vector, and each touch vector (finger gesture) gets its own arena. The global arena doesn't include recognizers; rather it includes a representation of each client's arena (a *contender*). Each client carries its *own* platform-distinct arenas and recognizers. The operation-system-specific logic for each platform is responsible for placing its arena's contender in the global arena.

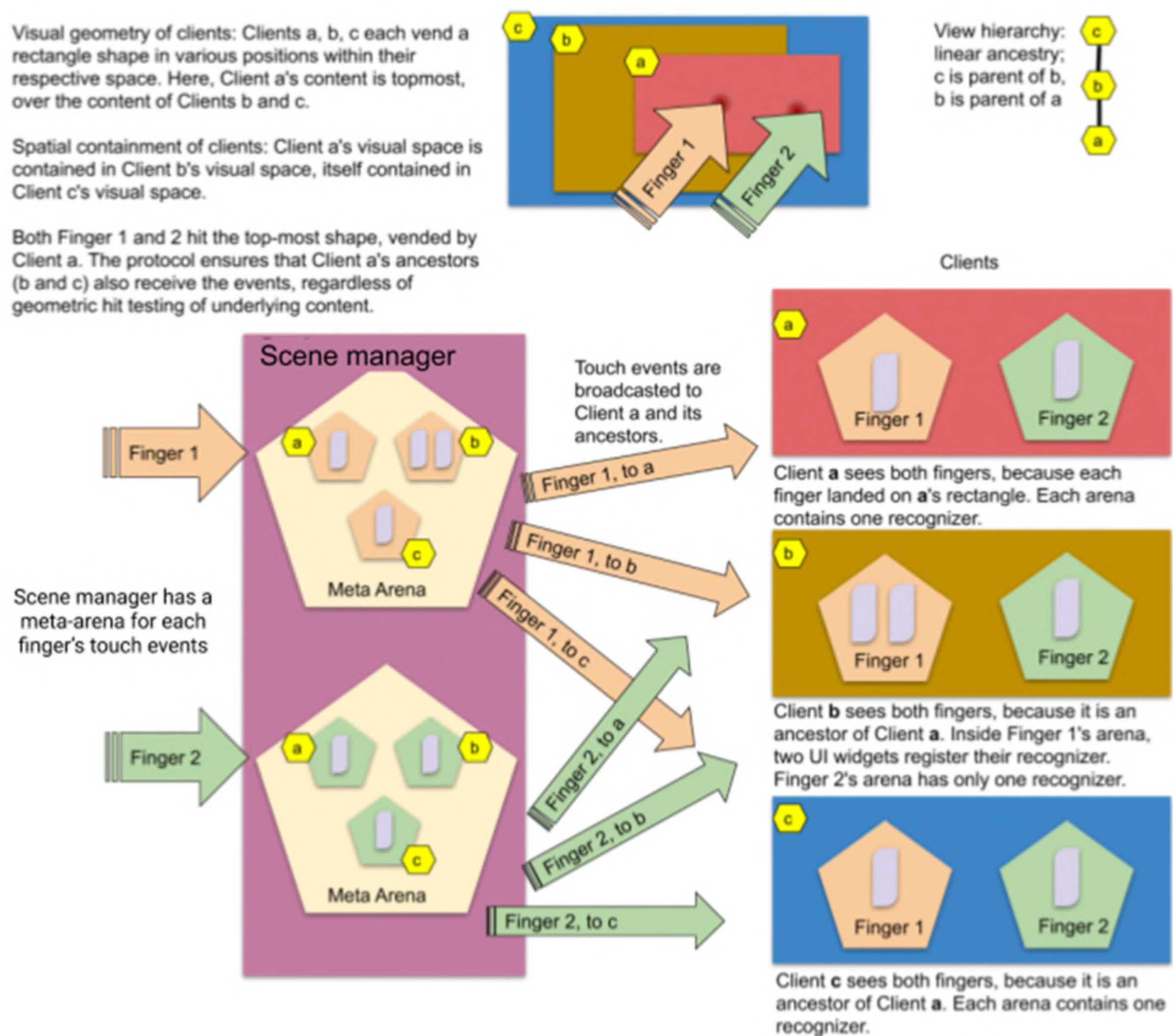

Fig. 2: Handling multi-finger gestures

Fig. 2 illustrates the handling of multi-finger gestures such as two-finger scroll. A single recognizer participates in multiple, parallel arenas, one for each finger. In Fig. 2, client *c* is the parent of client *b*, which is in turn the parent of client *a*. For example, client *c* can be a system UI, client *b* can be a shell program, while client *a* can be a mail application. Due to the visual geometry of the clients, a first finger hits all three clients, while a second finger hits clients *a* and *c*.

Gesture disambiguation is used to determine which client owns the touch event generated by the two fingers. As illustrated, the scene manager maintains a meta-arena for each finger, which broadcasts touch events corresponding to the fingers to the respective clients' arenas. For example, client *a* sees both fingers, one in each of two arenas that each include one recognizer. Client *b* sees just the first finger. Two UI widgets inside the arena of client *b* each register a recognizer. Client *c* sees both fingers, one in each of two arenas that each include one recognizer. Each recognizer sends back a signal indicating a claim on the touch event. The global arena adjudicates these claims and assigns the touch event generated by the two fingers to one of the clients.

Upon an ADD event, a scene manager determines a stable priority list of views to receive touch events. A *hit test* is a depth-first walk down a view hierarchy to collect the set of scene manager resources (shapes) that intersect the ADD event's ray projection. The hit test is sorted by distance (from the camera) to reveal the top-most resource along that ray projection, with its owning view V. The view V and all of its ancestors in the view hierarchy are the list of views, and that list is prioritized by depth in the tree, from least depth to greatest depth.

As soon as a touch event is received, the scene manager performs a *parallel dispatch* to all views in the priority list. The scene manager collects responses from each view, but does not

take action on those responses, until (a) it receives all of the requisite replies for each event, or (b) it determines that it doesn't need to wait for a reply, due to precedence, cancellation, or timeout. This enables decoupling the ANR timeout ("App Not Responding," usually a delay equivalent to one or two frames) from touch sampling frequency. Views that fail to respond in a timely manner are treated by the scene manager as having responded `No` to gestures.

Effectively, the scene manager view pipeline is a "serial-parallel" pipeline, where events E1, E2, E3, ... are dispatched ASAP to all views {a, b, ...} in the priority list, but further action on E1 is contingent on receiving responses R1-a, R1-b, ..., action on E2 is contingent on receiving responses R2-a, R2-b, ..., etc. Each view further dispatches the touch event to its internal set of gesture recognizers and integrates recognizer responses into a single arena status for reporting back to the scene manager.

Gesture protocol

Clients that participate in gesture disambiguation engage in the gesture protocol described herein. Clients that don't participate either declare as much in a view property, or respond with No to the `ADD` event. Clients that don't declare and don't respond are treated as ANR.

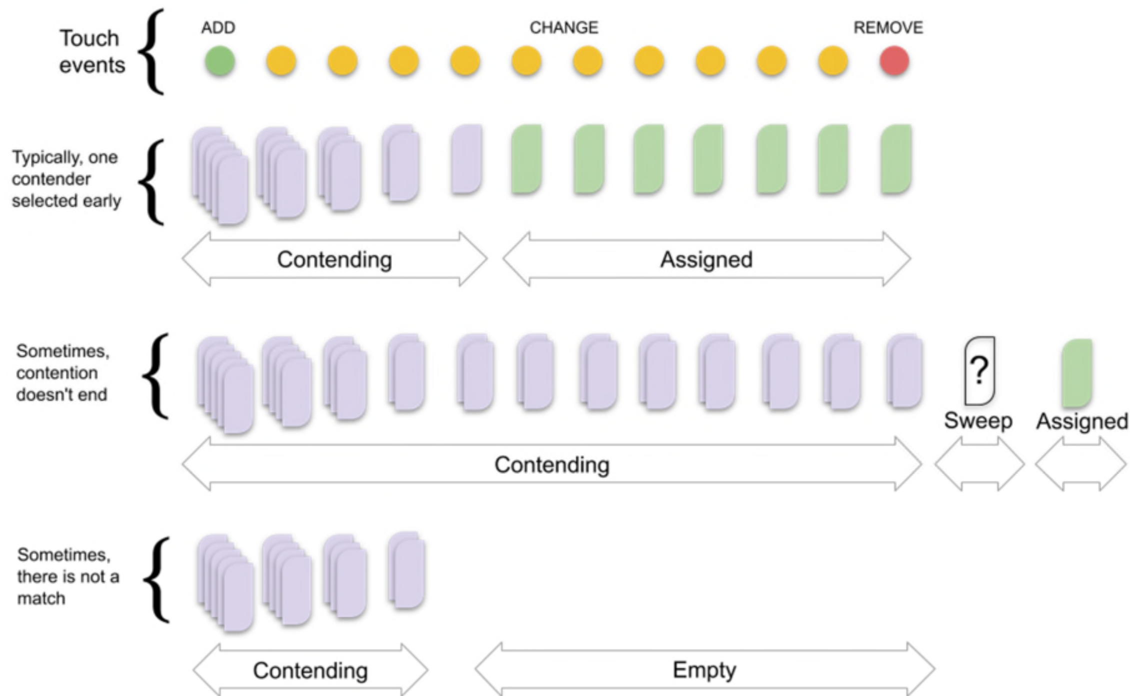

Fig. 3: The states of an arena

As illustrated in Fig. 3, the arena can have the following phases: {`Empty`, `Contending`, `Assigned`, `Sweep`}.

- `Empty` means there are no contenders in the arena. For example, the `ADD` event might not yield any gesture recognizers from any client. Another way to get `Empty` is for all contenders to reject the gesture.
- `Contending` means there are one or more contenders in the arena.
- `Assigned` means the arena has granted the win to a particular contender. Such a contender typically has a recognizer that has already responded `Yes`, but not necessarily.
- `Sweep` is a special case of `Contending`, where the arena has reached `REMOVE` without moving to `Assigned`.

  - The `Sweep` state may be prolonged by a `Hold`. After all relevant `Hold`s have transitioned to a non-`Hold` state, the `Sweep` may transition to `Assigned` or `Empty`.

Resolution is the process of moving to the Assigned phase. There are a few conditions where resolution is triggered:

- When contending, a contender is in the `Yes` state.
- When contending, there is a singleton non-`No`, non-`Yes` contender remaining.
  - This can happen on an earlier `CHANGE` event, but could also happen on the final `REMOVE` event.
- At `Sweep` (if resolution was not already triggered at `REMOVE`), all remaining contenders are non-`Yes`, non-`Hold`.

Once triggered, resolution immediately results in a single contender judged as the winner. As explained earlier, there are four basic contender states, {`Yes, No, Maybe, Hold`}, as follows.

- `Yes`: Contender asks to trigger resolution. Once a contender issues a Yes, the arena must declare the winner of the touch vector.
- `Maybe`: Contender continues contending, and is not ready to trigger resolution.
- `Hold`: Contender continues contending, and intends to consume a *subsequent* touch vector (e.g. look for a follow-up tap). The Hold state prevents a sweep from resolving.
  - A contender that outputs `Hold` typically should follow up with a transition to a non-hold state, e.g., `Yes` or `No`.
- `No` - Recognizer voluntarily declines from further contending.

When contending, a lower-priority `Yes` typically triggers resolution. At `sweep`, a lower-priority `Hold` typically prevents resolution. It is useful to have higher-priority contenders suppress these default behaviors; to do so, this disclosure introduces the attribute of suppress, notated as Suppress-Maybe ($\texttt{Maybe}_{s}$) and Suppress-Hold ($\texttt{Hold}_{s}$):

- $\texttt{Maybe}_{s}$: When Contending, prevents resolution by blocking a lower-priority `Yes`. At `Sweep`, triggers resolution by blocking a lower-priority `Hold`.
- $\texttt{Hold}_{s}$: When Contending, prevents resolution by blocking a lower-priority `Yes`. At `Sweep`, prevents resolution by blocking a lower-priority `Yes`.

The suppress attribute is "sticky", e.g., the presence of just one suppressive gesture in a contender effectively makes all the gestures from that contender suppressive. Suppression only works downward, e.g., a lower-priority suppressive state does not influence the interaction of higher-priority states. An example of a suppressive state is the accessibility client.

By default, a higher-priority winner yields its win to a lower-priority contender. To enable a higher-priority contender to prioritize their win upon resolution, this disclosure introduces the attribute of prioritize, notated as Prioritize-Yes ($\texttt{Yes}^{P}$) and Prioritize-Maybe ($\texttt{Maybe}^{P}$). To suppress a lower-priority state and prioritize a win, a state maybe-suppress-prioritize ($\texttt{Maybe}^{P}_{s}$) is introduced.

In this manner, the original basic four contender states are expanded to nine: $\{\texttt{No}, \texttt{Maybe}, \texttt{Maybe}^{P}, \texttt{Maybe}_{s}, \texttt{Maybe}^{P}_{s}, \texttt{Hold}, \texttt{Hold}_{s}, \texttt{Yes}, \texttt{Yes}^{P}\}$. The extra states ensure a wide range of contender precedence behavior.

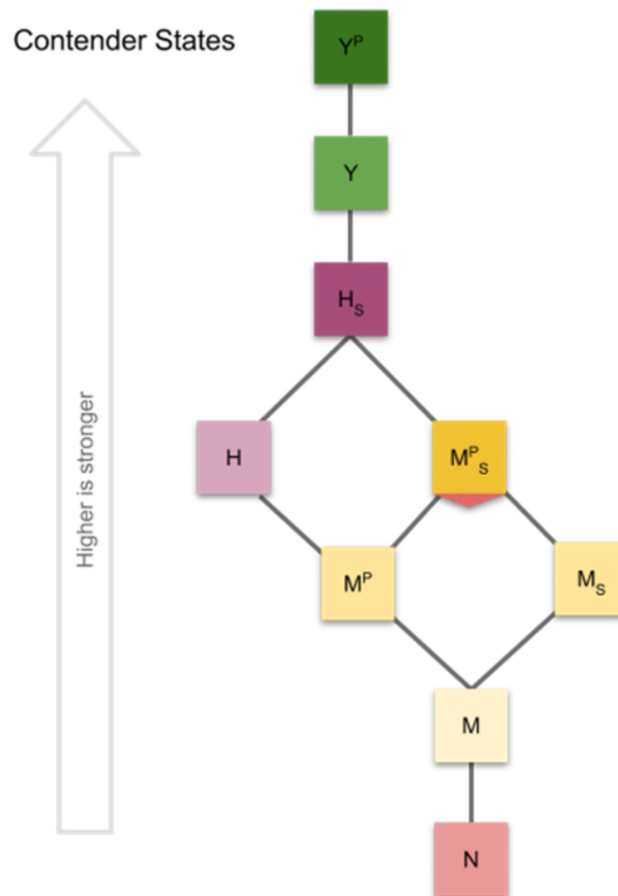

Fig. 4: A partial order of contender states

The above-described contender states form a *partial order* (lattice), as illustrated in Fig. 4. States higher in the lattice are considered stronger than those lower in the lattice. Some states are not directly relatable to each other, such as `Maybe`^{P} and `Maybe`$_S$. The lattice enables several gestures from a single client to be represented to the scene manager as a single node of the lattice, e.g., the nearest common higher node. For example, if the gesture recognizers of a client output states `Maybe`^{P} and `Maybe`$_S$, then the nearest common state, `Maybe`^P_S, is sent to the scene manager. As another example, if the gesture recognizers of a client output states `Hold` and `Maybe`$_S$, then the nearest common state, `Hold`$_S$, is sent to the scene manager.

The nearest common higher node is also known as the least upper bound, and is represented by a pairwise join operator ⊔. For example,

$$\texttt{Maybe}^{\texttt{P}} \sqcup \texttt{Maybe}_{\texttt{s}} = \texttt{Maybe}^{\texttt{P}}{}_{\texttt{s}},$$

$$\texttt{Hold} \sqcup \texttt{Maybe}_{\texttt{s}} = \texttt{Hold}_{\texttt{s}}, \text{etc.}$$

The join operator ⊔ correctly represents the state for a priority list of contenders. That is, if an arena consists of N recognizers, each with some contender state $\{c_1, \ldots, c_N\}$, then the combined state of that arena is

$c_1 \sqcup \ldots \sqcup c_N$.

Due to this property of the join operator, synthetic states such as the non-existent $\texttt{Hold}^{\texttt{P}}{}_{\texttt{s}}$ are obviated. For example, to account for the combination of $\{\texttt{Hold}, \texttt{Maybe}^{\texttt{P}}{}_{\texttt{s}}\}$, the join element, $\texttt{Hold}_{\texttt{s}}$, encodes sufficient details to correctly represent this set of recognizers during all phases of the arena, including resolution.

After a contender responds to an input event, the scene manager can follow up with one of the following:

- OwnershipGranted: This is the scene manager's confirmation bit, which indicates that the contender has been granted the win.
    - If the contender is representing a `Yes` recognizer, the contender is obligated to give that recognizer the confirmation bit. If there are multiple `Yes`es, it should immediately assign one of them as the winner.
    - If the contender has only `Maybe` or `Hold` recognizers, it should eventually assign one of them as the winner.
- OwnershipRejected: The contender is not granted the win and is obligated to terminate all its recognizers.

The response vector of the arena

In the scene-manager arena, the ADD event generates a priority list **L** of contenders, each from a view. For each broadcast of ADD and CHANGE events to all contenders in **L**, a response is expected from all contenders in **L**, even if it is to decline participation. The ordered list of responses is a *response vector*. Each response vector **R** should evaluate to a deterministic action, where the order of elements in **R** directly reflects priority order in **L**. To do so, the following *sequencing operator* is defined over contender states, where "a ; b" means "a precedes b" in **R**. The precede operator ";" is defined thus: For all e in { No, Maybe, Maybe^{P}, Maybe_{s}, Maybe^{P}_{s}, Hold, Hold_{s}, Yes, Yes^{P} },

$$\mathrm{Yes}^{P} \;;\; e = \mathrm{Yes}^{P}$$

$$\mathrm{Yes} \;;\; e = \mathrm{Yes}$$

$$\mathrm{Hold}_{s} \;;\; e = \mathrm{Hold}_{s}$$

$$\mathrm{Hold} \;;\; e = \mathrm{Hold} \sqcup e$$

$$\mathrm{Maybe}^{P}_{s} \;;\; e = \mathrm{Maybe}^{P}_{s}$$

$$\mathrm{Maybe}_{s} \;;\; e = \mathrm{Maybe}_{s}$$

$$\mathrm{Maybe}^{P} \;;\; e = \mathrm{Maybe}^{P} \sqcup e$$

$$\mathrm{Maybe} \;;\; e = \mathrm{Maybe} \sqcup e$$

$$\mathrm{No} \;;\; e = \mathrm{No} \sqcup e \quad = e$$

Then the response vector $(r_0, r_1, \ldots, r_n)$ evaluates to the following result:

$$r = r_0 \;;\; r_1 \;;\; \ldots \;;\; r_n$$

The result of an empty vector is defined to be No.

The sequencing operator is associative. The suppressive states can prevent resolution triggered by lower-priority states; the deferential (default) states enable it by use of the join operator.

<u>*Actions on Response Vector and Resolution*</u>

The arena's action for each touch event is dictated by the result of the response vector. All `No` responses are removed from the vector such that remaining contenders are guaranteed non-`No`. Then the arena acts in one of the following ways; each is the result of the entire response vector rather than an individual contender.

- `No`: all contenders have responded `No`. Move to the `Empty` phase.
- `Maybe`
    - If `Contending` and single, then resolve as the winner. If `Contending` and multiple, keep going (e.g., there is as yet insufficient information about the touch event).
    - If at `Sweep`, then resolve in favor of the lowest-priority `Maybe`. (Lower priority states are guaranteed to be less than or equal to `Maybe`.)
- $\texttt{Maybe}^{P}$
    - If `Contending` and single, then resolve as the winner. If `Contending` and multiple, keep going (e.g., there is as yet insufficient information about the touch event).
    - If at `Sweep`, then resolve in favor of the highest-priority $\texttt{Maybe}^{P}$. (Lower priority states are guaranteed to be less than or equal to $\texttt{Maybe}^{P}$.)
- $\texttt{Maybe}_{S}$

  - If `Contending` and single, then resolve as the winner. If `Contending` and multiple, keep going (e.g., there is as yet insufficient information about the touch event).
  - If at `Sweep`, then drop `Hold`s lower than the highest `Maybe`$_{s}$, and resolve in favor of the highest-priority `Yes`^{P}, the lowest-priority `Yes`, the highest-priority `Maybe`^{P}, or the lowest-priority `Maybe`.
- `Maybe`$^{P}_{s}$
  - If `Contending` and single, then resolve as the winner. If `Contending` and multiple, keep going (e.g., there is as yet insufficient information about the touch event).
  - If at `Sweep`, then drop `Hold`s lower than the highest `Maybe`$^{P}_{s}$, and resolve in favor of the highest-priority `Yes`^{P}, the lowest-priority `Yes`, or the highest-priority `Maybe`^{P}.
- `Hold`
  - If `Contending` and single, then resolve as the winner. If `Contending` and multiple, keep going (e.g., there is as yet insufficient information about the touch event).
  - If at `Sweep`, prevent resolution. (Lower priority states are guaranteed to be less than or equal to `Hold`.)
- `Hold`$_{s}$
  - If `Contending` and single, then resolve as the winner. If `Contending` and multiple, keep going (e.g., there is as yet insufficient information about the touch event).

  - If at `Sweep`, then ignore any `Yes` lower than the highest $\texttt{Hold}_S$, and prevent resolution.
- `Yes`
  - At `Contending` and at `Sweep`, resolve in favor of the highest-priority $\texttt{Yes}^P$, or the lowest-priority `Yes`. (Lower priority states may be suppressing an even lower-priority $\texttt{Yes}^P$ or `Yes`).
- $\texttt{Yes}^P$
  - At `Contending` and at `Sweep`, resolve in favor of the highest-priority $\texttt{Yes}^P$.

Advantages of the gesture protocol

The described gesture protocol has several advantages, including:

- Determinism in the absence of app-not-responding (ANR) condition: When no contender drops out due to ANR, and when each contender's arena responds deterministically to a given touch event, then the result of the response vector is also deterministic.
- Priority-based disambiguation: The result of the response vector is determined by the order of its constituent responses. In particular, touch events are dispatched in parallel to the recognizers, and suppression works to prevent action by lower-priority contenders while still enabling the participation of lower-priority gestures.
- Prevention of denial of service: The absence of response from a lower-priority contender may temporarily delay event processing up to a predefined timeout, but treating it as ANR takes it out of consideration for future events.
- Enabling of custom recognizers: The presence of developer-authored custom recognizers (e.g., an A-shaped swipe) means a centralized and vetted set of gesture recognizers cannot be assumed. The described gesture disambiguation protocol enables clients to

perform arbitrary behaviors without losing stability or incurring undue latency. In particular, the suppress and prioritize attributes enable higher-priority contenders to override or ignore responses from lower-priority clients.

It is conceivable that a very high-priority client, e.g., an accessibility client, retains a tap gesture for special work and only passes through double-taps as a regular tap. In other words, regular clients are to be sent a tap touch vector when the accessibility service recognizes a double-tap, making gesture disambiguation impractical. Per the techniques described herein, there is no gesture reinterpretation prior to parallel dispatch. All clients agree on the nature of a gesture (e.g., a tap is a tap for all clients). High-priority clients like accessibility implement high-priority suppress-and-override to generate semantic actions. The described approach avoids the error-prone nature of synthesizing touch vectors to cause specific actions (e.g., synthesize a tap to click a button).

CONCLUSION

This disclosure describes techniques to disambiguate user gestures made on a touchscreen device which can be incorporated in an operating system. The techniques enable simultaneously-running clients to define and claim gestures; provide a centralized mechanism to deny gestures; provide an expression of priority amongst competing clients; provide a mechanism for a parent client to defer a gesture to a child client; etc. Simultaneous access of a gesture by competing clients eliminates latency across the full breadth of client priorities.