Flutter
Trackpad Gestures

SUMMARY

Trigger GestureRecognizers using trackpad gestures.

Author: Callum Moffat (moffatman)
Go Link: flutter.dev/go/trackpad-gestures
Created: 08-2021 / **Last updated:** 01-2022

OBJECTIVE

- Flutter engine should use platform APIs to recognize trackpad gestures
- Flutter engine should pass trackpad gestures to the framework statefully
- `GestureRecognizers` in the framework should react to those gestures

BACKGROUND

Due to Flutter's focus on mobile platforms, apps have been designed for a wide range of rich touch gestures. The recent addition of desktop platform support means some apps might have interactions which are awkward with a mouse and keyboard. For example, dismissing an image by dragging it to the side, or swiping between pages. Using a pointer to directly manipulate content is very intuitive on a touch screen, but clicking-and-dragging with a mouse to perform that same interaction is not intuitive. Proper implementation of trackpad gestures will greatly improve the experience of those interactions.

Rich trackpad support is something receiving expanded support recently, with new efforts from Microsoft to improve the trackpad experience on Windows. Flutter apps should support trackpad gestures to offer feature parity with other ways of developing applications.

Glossary

- **Pan Gesture**
  - Action: Two fingers moving together on a trackpad
  - Intention: Move/translate the displayed content
- **Zoom Gesture**
  - Action: Two fingers moving farther apart or closer together
  - Intention: Increase or decrease the scale of the displayed content
- **Rotate Gesture**
  - Action: Two fingers rotating around a central point
  - Intention: Rotate the displayed content

OVERVIEW

The existing system for handling desktop scroll events (`PointerSignal`) works well for handling stateless events such as those generated by a desktop mouse. Each scroll wheel increment can be treated as a separate event and handled by different widgets without consideration of previous events. But events as part of a trackpad gesture should behave differently.

Each event of the same gesture should be handled by the same widget. Interactions such as pull-to-refresh and flinging that are common in Flutter apps rely on the timing of when a pan interaction starts and ends. So we need to communicate not just the intent of the gestures (moving or zooming the content view), but when they begin and finish.

Three new types of pointer events will be introduced: `PointerPanZoomStartEvent`, `PointerPanZoomUpdateEvent`, and `PointerPanZoomEndEvent`. `PointerPanZoomStartEvent` and `PointerPanZoomEndEvent` will not contain any additional data, but simply indicate to the framework that a gesture is starting or a gesture has ended. `PointerPanZoomUpdateEvent` will contain some additional fields to represent a combination of pan, zoom, and rotate gestures.

```
class PointerPanZoomUpdateEvent extends PointerEvent... {
 /// The total pan offset of the gesture
 final Offset pan;
 /// The amount the pan offset changed since the last event
 final Offset panDelta;
 /// The scale (zoom factor) of the gesture
 final double scale;
 /// The amount the gesture has rotated in radians so far
 final double angle;

 /// ...
}
```

Using a single type of gesture event will allow `ScaleGestureRecognizers` to compete with `DragGestureRecognizers`. Even though most platforms lock a gesture to a single type of interaction (such as pan vs zoom) before sending it to applications, there are some which allow multiple types of gesture at once.

Non-goals

- A better experience for mice with scroll wheels

DETAILED DESIGN/DISCUSSION

Gesture data availability

Legend

Official support from OS	Possible with caveats	Not needed	Not possible, but still usable	Not possible

Data

	iPadOS	macOS	ChromeOS	Android	Windows	Linux	Web
Pan (scroll)	✔	✔	✔	X	✔	✔	X
Zoom	✔	✔	X	X	✔	✔	X
Rotate	✔	✔	X	X	X	✔	X
Raw Trackpad Data	X	✔	X	✔	X	✔	X

iPadOS
First-party APIs are available in UIKit to receive well-formed streams of gesture data for pan, zoom, and rotate.

macOS
First-party APIs are available in AppKit to receive well-formed streams of gesture data for pan, zoom, and rotate.

ChromeOS
The Android runtime for ChromeOS simulates a pointer when the user scrolls on a trackpad, since that point has unique properties (buttonState=0), we can intercept it and turn it into a gesture. The advantage versus remaining with the simulated pointer is that it won't trigger any taps or other inappropriate recognizers. When a user performs a pinch-to-zoom or rotate gesture, ChromeOS translates it directly into multiple touches to the application. There isn't a reliable way to differentiate

those pointers from legitimate touchscreen presses, so we can't handle them properly.

Android
There are very few devices with a trackpad that run vanilla Android and not ChromeOS. According to documentation the raw gesture information is available, but the engine will need to be extended to turn the raw trackpad touch points into gesture data. A test of a trackpad plugged into an Android tablet showed only mouse movement was handled by the system. Scrolling did not work in any app, so this platform is not a priority, it might be better to wait for system APIs to improve instead of implementing custom event logic.

Windows
The DirectManipulation API on Windows can be used to receive a combined pan and zoom transform. According to documentation, it should be possible to get rotation as well, but it doesn't happen in practice.

This is the API used by Chrome and Firefox. It only seems to work using the more modern types of trackpad called "Precision Touchpad", since those use a first-party driver from Microsoft. Older trackpads from vendors such as Synaptics use custom drivers which present to applications as mice with scroll wheels, and strip out the vital timing of when gestures start and finish.

Linux
First-party APIs are available in GTK to receive well-formed streams of gesture data for pan, zoom, and rotate.
Caveat: Environment variable GDK_BACKEND needs to be unset when running in Wayland. If XWayland is used, the gesture data will not be available.

Web
Most web browsers don't support handling trackpad gestures by the website. Chrome and Firefox use the user's trackpad gestures to manipulate the webpage viewport directly. The events that make it to the JavaScript runtime are missing the timing of when gestures start and end. There isn't a way to get gesture events beyond stateless scroll wheel increments. Safari has a non-standard Gestures feature which could allow implementation for WebKit-based browsers. Discussion here.

Fuchsia
I have no knowledge of this platform. It appears upon cursory investigation that trackpad gestures are still in the RFC phase in Fuchsia.

Internal API Changes

The data for `PointerPanZoomUpdateEvent` will come from a few new fields in pointer data packets.

```
struct alignas(8) PointerData {
  // Must match the PointerChange enum in pointer.dart.
  enum class Change : int64_t {
    // ...
    kGestureDown,
    kGestureMove,
    kGestureUp,
  };

  // ...
  double pan_x;
  double pan_y;
  double pan_delta_x;
  double pan_delta_y;
  double scale;
  double angle;

  // ...
};
```

Straightforward plumbing changes will be needed within `GestureBinding` and `PointerEventConverter` to wire up the events.

GestureRecognizer

Right now, new pointer sequences are added to `GestureRecognizers` by calling `void addPointer(PointerDownEvent event)`. Since trackpad gestures will start with a `PointerPanZoomStartEvent` instead, the API for `GestureRecognizer` will need to have new methods to accept those events.

```
abstract class GestureRecognizer extends GestureArenaMember with
DiagnosticableTreeMixin {
 /// ...
 void addPointerPanZoom(PointerPanZoomStartEvent event) {
   _pointerToKind[event.pointer] = event.kind;
   if (isPointerPanZoomAllowed(event)) {
     addAllowedPointerPanZoom(event);
   } else {
     handleNonAllowedPointerPanZoom(event);
   }
 }
```

```
@protected
void addAllowedPointerPanZoom(PointerPanZoomStartEvent event) { }

@protected
void handleNonAllowedPointerPanZoom(PointerPanZoomStartEvent event) { }

@protected
bool isPointerPanZoomAllowed(PointerPanZoomStartEvent event) {
  return false;
}
}
```

These will only need to be overridden in `GestureRecognizers` that want to recognize trackpad gestures, such as `DragGestureRecognizer` and `ScaleGestureRecognizer`. Other gesture recognizers don't need to hear anything about trackpad gestures, so they won't receive any events.

Why should trackpad gestures be part of PointerEvent/GestureArena, rather than PointerSignalResolver?

Trackpad gestures share almost all of the characteristics of touch pointers that are solved by the `GestureArena` system. They need to be hit-tested and routed, and need to be claimed by one recognizer amongst multiple possible options. This is because a `PointerPanZoom` gesture contains all the degrees of freedom as two touch points, representing a combination of pan, zoom, and rotate. The exact meaning in each scenario will be resolved by the competition amongst different `GestureRecognizers`. We need to use the `GestureArena` system to pick whether a trackpad gesture should cause a swipe between gallery pages, or a zoom onto the current gallery image based on the characteristics of the gesture (pan distance, zoom scale, time duration). Using the `GestureRecognizer` system lets those decisions use existing mechanisms for that work, and drive existing apps with rich gesture support.

Why change the current GestureRecognizer classes?

`GestureRecognizers` have maintained an invariant - they track only one gesture at once, so you will always get a sequence such as (`scaleStart`, `scaleUpdate`, `scaleEnd`) before the next gesture starts. The solution for trackpad gestures needs to ensure that apps only receive one gesture at once, even if both the trackpad and touch screen are receiving input. Adding support for pan/zoom events to the `GestureRecognizer` superclass and combining gesture inputs in the relevant existing `GestureRecognizers` is the most straightforward way to do this.

Why can't we split GestureRecognizer for touch v.s. trackpad?

A potential alternative design might have a new class to handle recognition of `PointerPanZoom` events. The task of merging potential recognized gestures into a maximum of one concurrent event could be done by some new `GestureRecognizerCombiner`. For example, the default `DragGestureRecognizer` might become a composition of a `DragTouchGestureRecognizer` and a `DragPanZoomGestureRecognizer`. This approach is not possible, though, as there is not a strictly one-way data flow from pointer data to gesture callbacks. The recognition and disambiguation process depends on the current state of each gesture recognizer. For example, an active scale gesture will immediately claim any new pointers it is notified of. If trackpad recognition is done in a different class, it won't know about that, and a trackpad gesture might be claimed by the scale widget's scrolling viewport by mistake.
On the other hand, most platforms do not handle simultaneous touch and trackpad very well at all, so if we accept inconsistent behavior when that is attempted, not all these points need to be followed, and maybe `GestureRecognizer` could be split. But a splitting strategy has some of its own problems (for example, customizations to gestures will require modifications to three classes instead of one).

Listener

`Listener` will be extended with new callbacks for the new types of pointer events.

```
class Listener extends SingleChildRenderObjectWidget {
 /// Creates a widget that forwards point events to callbacks.
 const Listener({
   ...
   this.onPointerPanZoomStart,
   this.onPointerPanZoomUpdate,
   this.onPointerPanZoomEnd,
 });

 /// ...

 /// Called when a gesture begins such as a trackpad gesture
 final PointerPanZoomStartEventListener? onPointerPanZoomStart;

 /// Called when a gesture is updated
 final PointerPanZoomUpdateEventListener? onPointerPanZoomUpdate;

 /// Called when a gesture finishes
 final PointerPanZoomEventListener? onPointerPanZoomEnd;
}
```

Public API Changes

For anyone simply using `GestureRecognizers` or `GestureDetectors` , there won't be any changes at all. Those recognizers will start emitting gesture callbacks based on events from the trackpad. Custom handling could be done with `Listener`'s new callbacks.

Other details

Inertia / Smooth Scrolling

It is up to the users of `GestureRecognizers` to decide what happens after the user finishes their input. `Scrollable`, for example, decelerates the results of its gestures to zero, scrolling smoothly to a stop. But other widgets change state right away when the gesture completes. For example, `PageView` animates the end of a gesture by snapping to the nearest page. To maintain these behaviors, it's important that the gesture events sent by the engine end the moment the user lifts their fingers off the trackpad.

At the end of a gesture, many platforms create additional scroll events to smoothly decelerate. Luckily, in all supported platforms we are able to disambiguate and pass through only real gesture events, and ignore any ones the system synthesizes. Using the platform-synthesized scroll events could allow Flutter apps to have identical scroll behavior to other applications on the system, but that data needs to be provided through a separate channel, and not `GestureRecognizers`. For this change, Flutter framework will be the only place that scroll inertia is generated. A way to send platform scroll inertia to interested widgets such as `Scrollable` should be a separate change.

Impact on engine performance

Adding these new fields will increase the size of each pointer packet by 48 bytes, which is an increase in memory copy of 5.7KB/s assuming a 120Hz sample rate. A simple optimization (sharing the fields for `scrollDelta` and `panDelta`) could reduce that to +32B / +3.8KB/s. A more complex rewrite of the pointer packet using a union of structs to reduce the number of total fields sent could result in an overall net effect of -16B / -1.9KB/s.

The baseline data transfer from pointer packets using the same assumptions is 27.8KB/s.

OPEN QUESTIONS

- ScaleSlop is based on absolute pixels, but the scale value in `PointerPanZoomUpdateEvent` will be relative. I made up 1.05 "scale units" instead of 18 pixels to trigger the `ScaleGestureRecognizer` . What should this value be?
- Is special handling needed to forward gestures to embedded PlatformViews?

  Since PlatformViews are not merged in any desktop embedding yet, I think this issue hasn't come up.
- What should the value for pointerCount be in `ScaleUpdateDetails` callbacks?
- Should we change `DragUpdateDetails` and `ScaleUpdateDetails` to indicate the gesture came from a trackpad?
- ~~Come up with a better name than "pointer gesture", it causes some confusion in the `GestureRecognizer` code~~ ~~Name change is to Pointer Flow~~ Name change is to PointerPanZoom

TESTING PLAN

Unit tests in the framework to ensure that platform gesture events correctly trigger GestureRecognizers. Also need to ensure that multiple gestures at once as well as gestures + touches will be recognized correctly.

Only iOS/macOS have any support for triggering trackpad gestures as part of end-to-end testing.

MIGRATION PLAN

This should not break any existing workflows or APIs. Some gesture events will be sent instead of scroll events, but unless an app has been designed solely for desktop without using any `GestureRecognizers`, the experience will be improved automatically. Widgets that use `GestureRecognizers` directly instead of `GestureDetectors` will need to forward the `onPointerPanZoomStart` callback of `Listener` to their `GestureRecognizers` to have them respond to trackpad input. iPadOS apps will need to set a new key in Info.plist to get zoom and rotate gestures through this system, but until that key is set, iPadOS will emulate the gestures by faking touch points. So this isn't a breaking change for iPadOS.