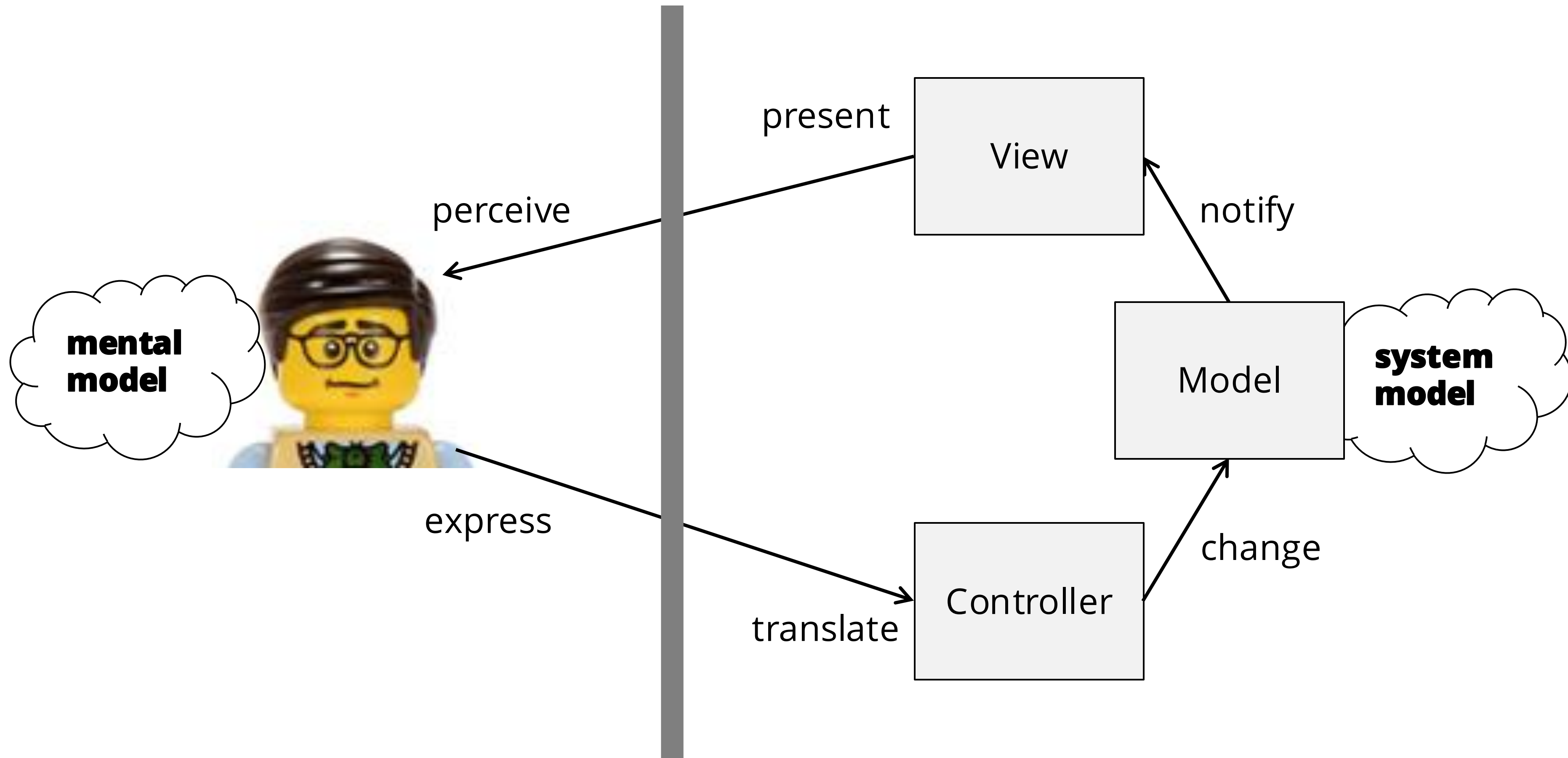
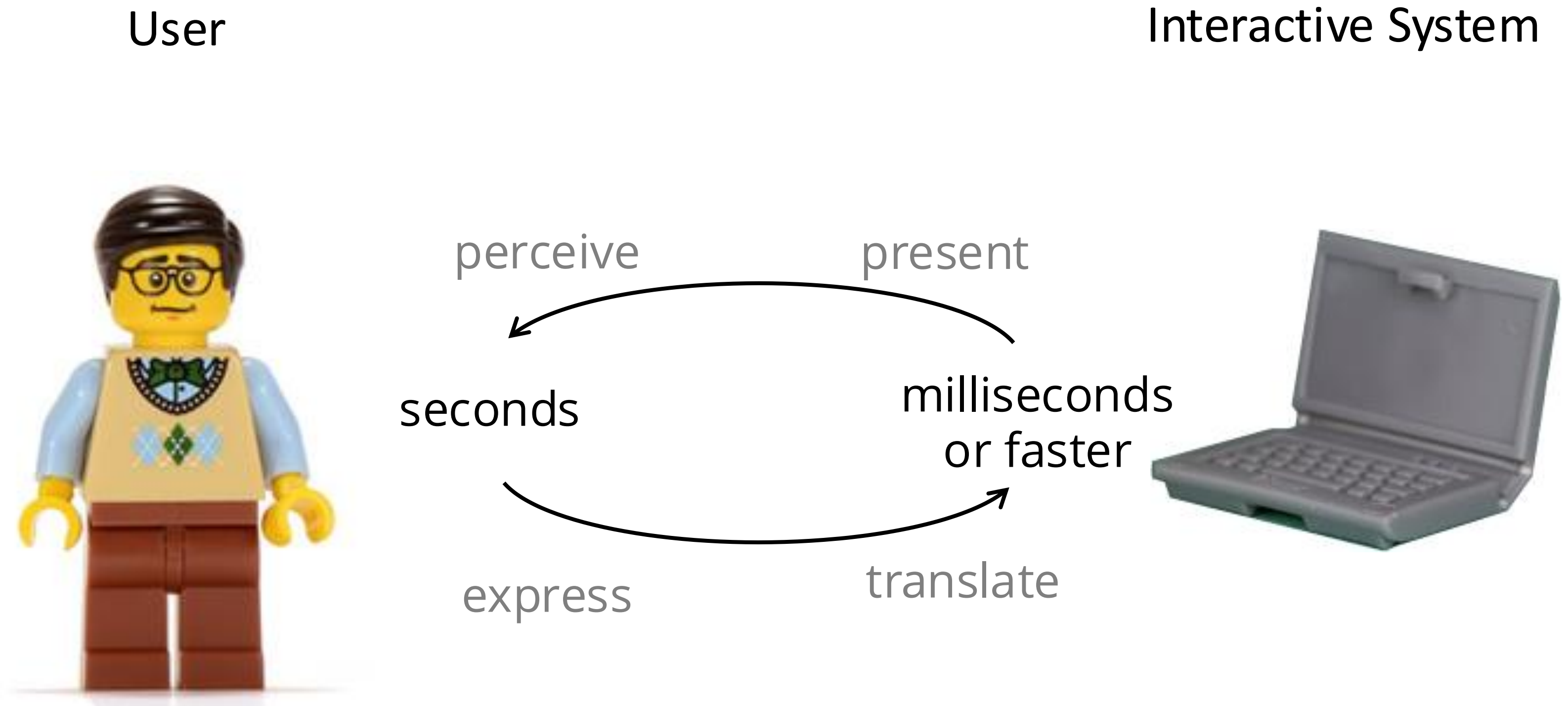
Input Events

- Event-driven programming
- OS event loop
- Toolkit run loop
- Event translation

Model-View-Controller (MVC)



Event Driven Programming



Event-driven programming is a programming paradigm that bases program execution flow on *events*. These can represent user actions or other system actions.

Event

- *In general English usage:*
  - An observable occurrence, often extraordinary occurrence
- *In user interface architecture:*
  - A message to notify an application that something happened

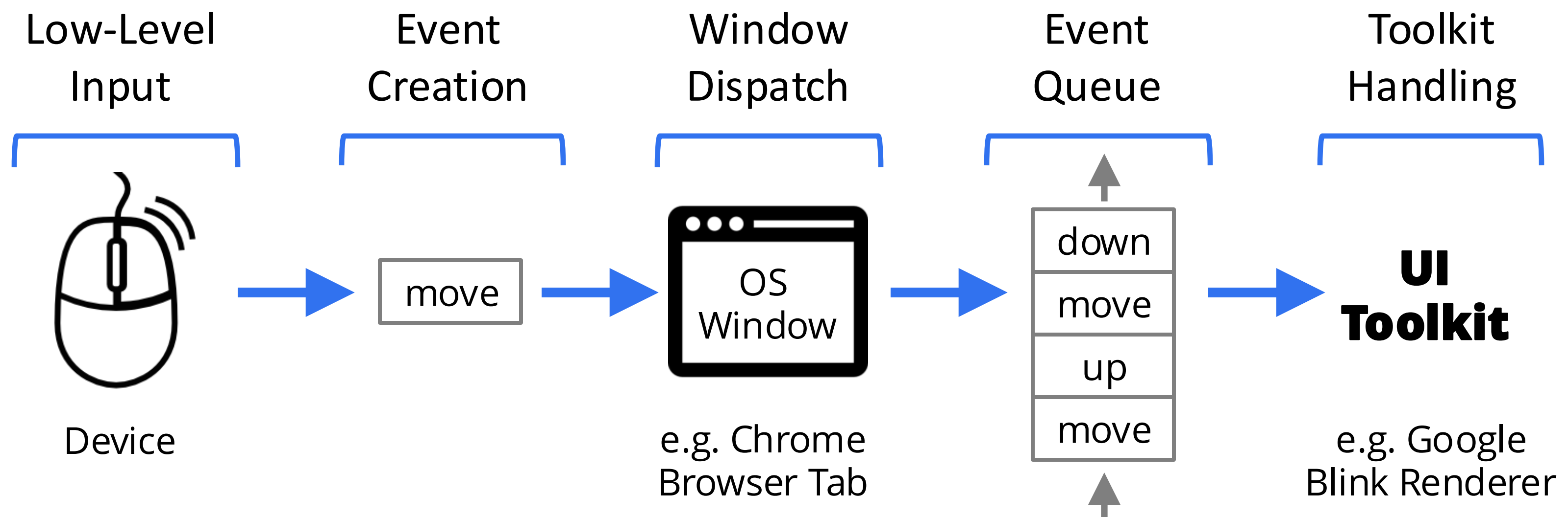
Event Types

- **Device Input Events**
  - Keyboard (e.g. key press, key release)
  - Pointing (e.g. move move, button press, button release)
- **Window Input Events**
  - Changes (e.g. resize, closing)
- **Window or Widget Events**
  - Pointing (e.g. mouse enter, mouse leave)
  - Focus (e.g. focus gained, focus lost)
- **System Events**
  - Timer (e.g. tick, completed)
- **Application Events**
  - Thread (e.g. progress, completed, ...)

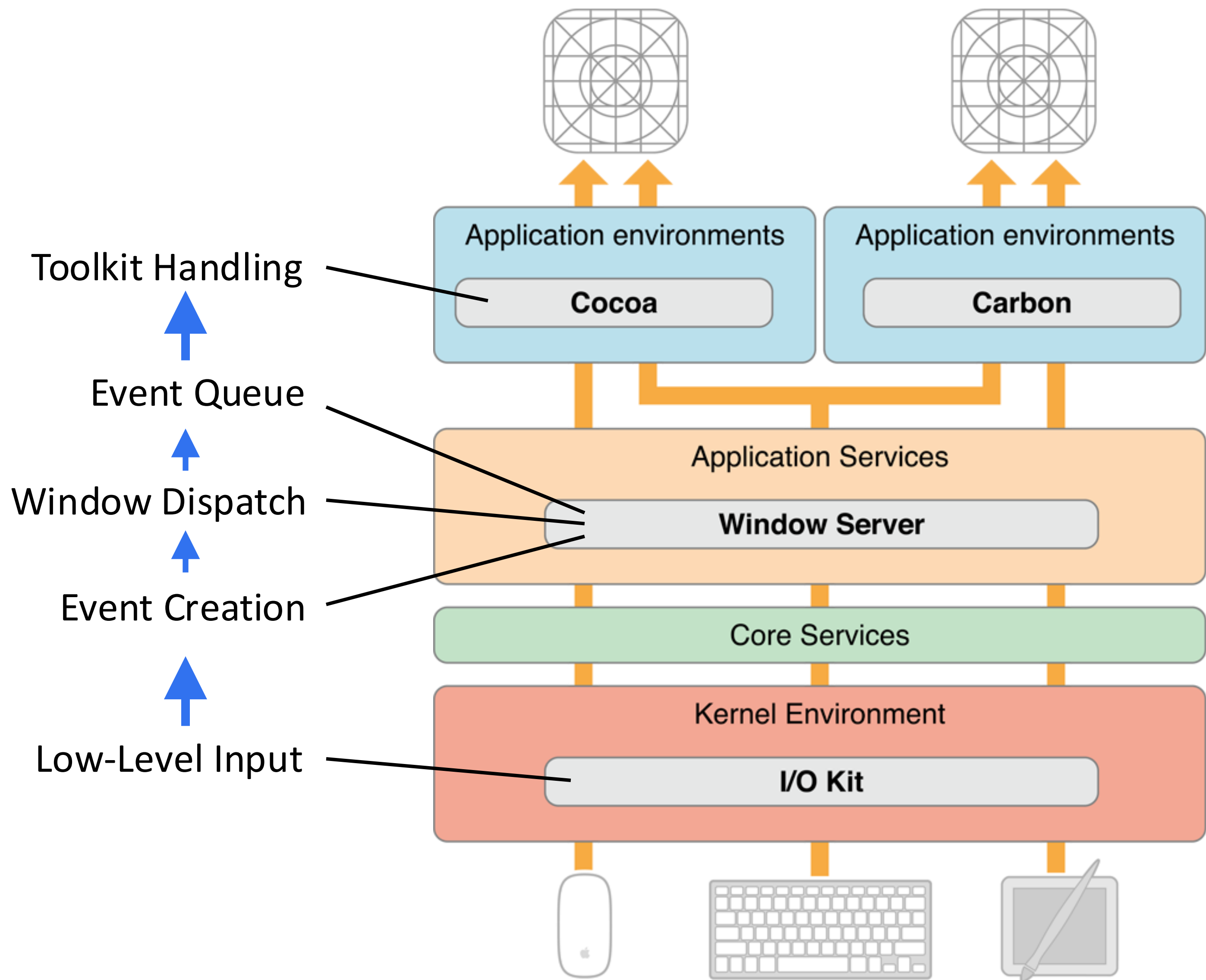
Input Event Architecture

The OS polls the input device state and communicates changes in the form of events to the "active" application window

The process can be modelled as a pipeline:



Example: OSX Event Architecture

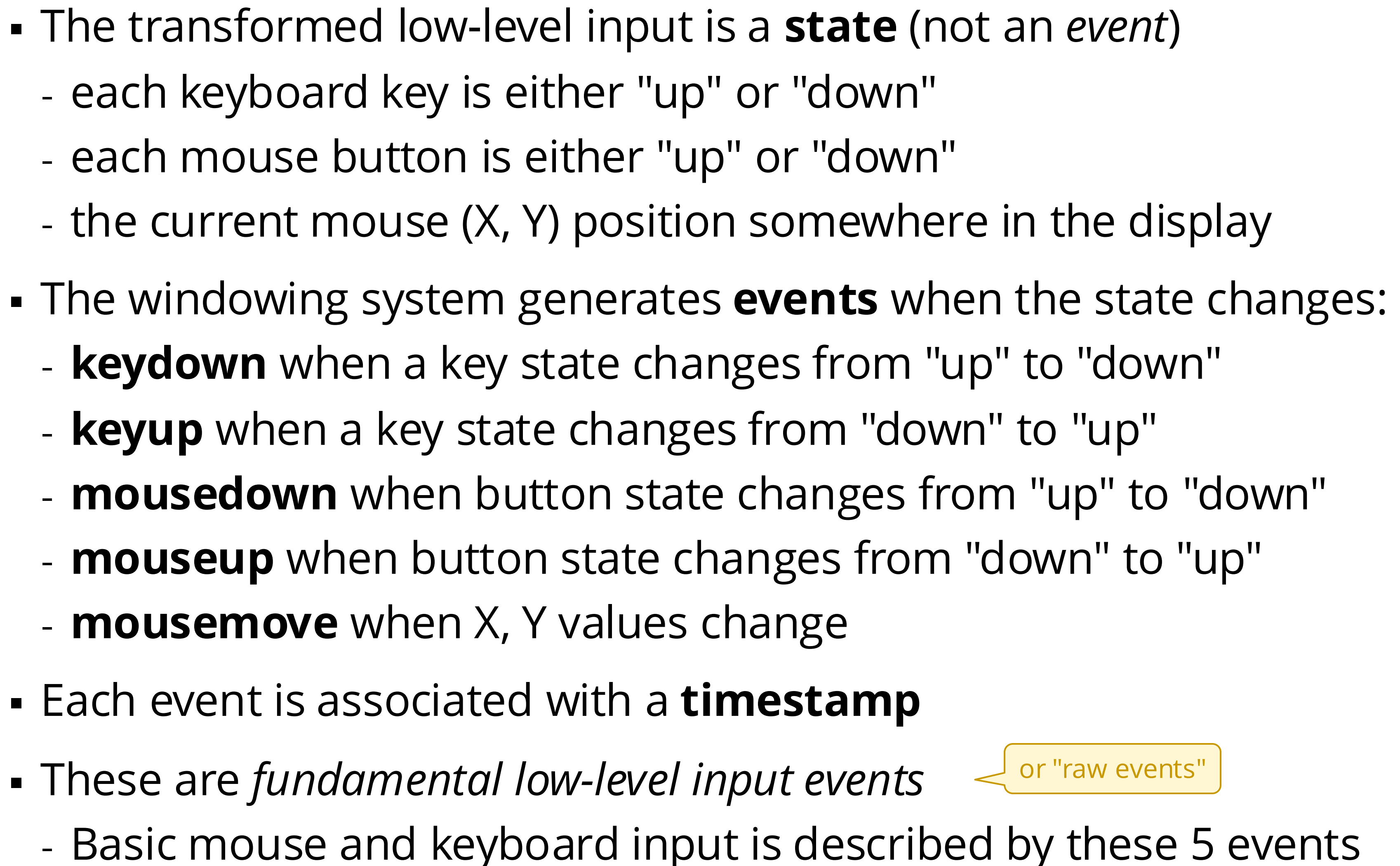


https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/EventOverview/EventArchitecture/EventArchitecture.html

Low-Level Input

- Most mice and keyboards conform to the Human Interface Devices (HID) standard
  - each device reports to OS what data will be sent (called a "boot report format")
  - *Example data for a mouse:*
    2 bytes for X and Y relative movement, each [-127, 127] "counts"
    1 byte button with state (button 1, 2, 3 with down/up)
- The OS **polls** the device to get current state
  - typically, every 8ms (125 Hz)
- The OS filters and transforms input data
  e.g. for mouse input:
  - convert relative X, Y "counts" into a velocity
  - apply a "pointer acceleration function" to adjust velocity
  - use velocity to move mouse cursor in display coordinates

Event Creation

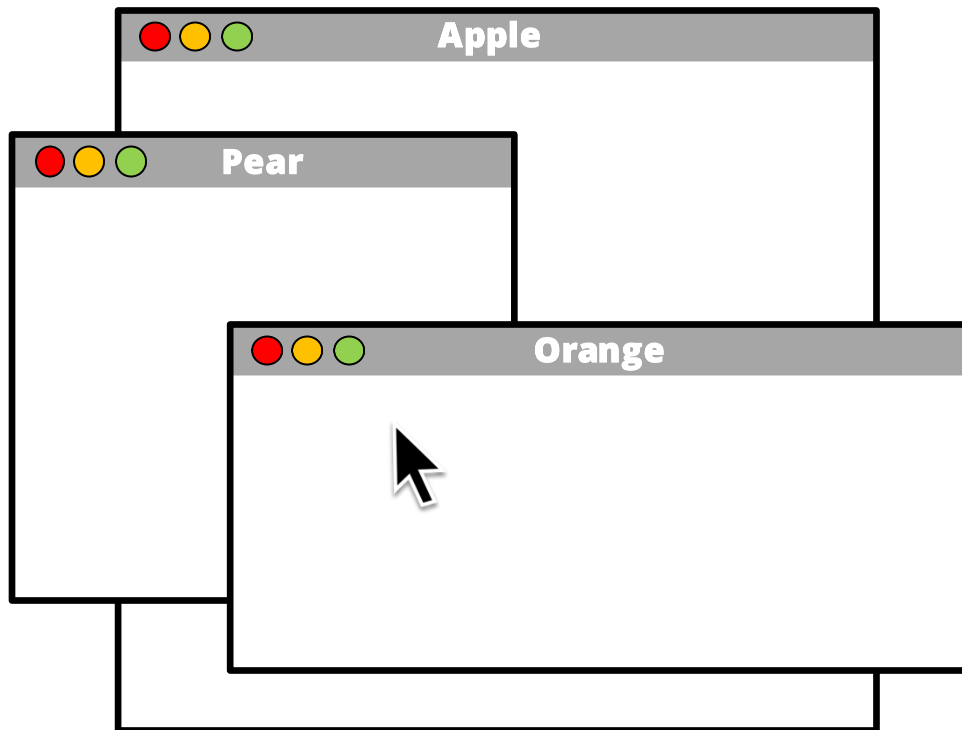


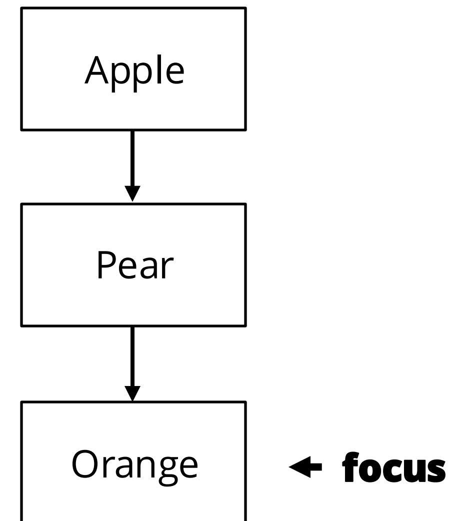
Window Dispatch

like a display list

- The windowing system maintains a list of all windows ordered from back to front, the front most window has *focus*
  - events* are sent to focused window (e.g. to the UI Toolkit)

*some exceptions: global hooks, overlays, ...





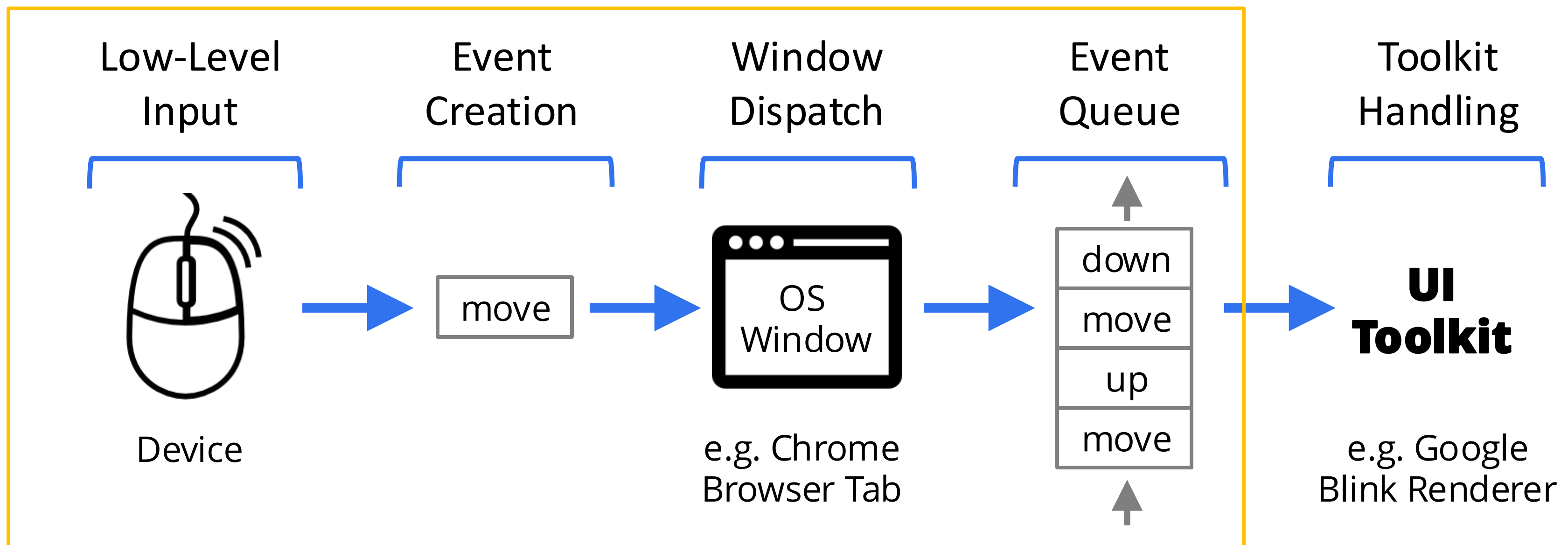
Event Queue

- The event queue is a buffer between the user and each window
- User input events tend to be "bursty"
  - several seconds pass with none, then many in quick succession
- Queuing lets UI toolkit running in window handle events efficiently
  - can be some *delay* before handling an event
    ... but not too much, or input "lag" is introduced
- Toolkit should **refer to event timestamp**, *not time when the event was pulled off the queue or handled by the application code*

Windowing System Event Loop

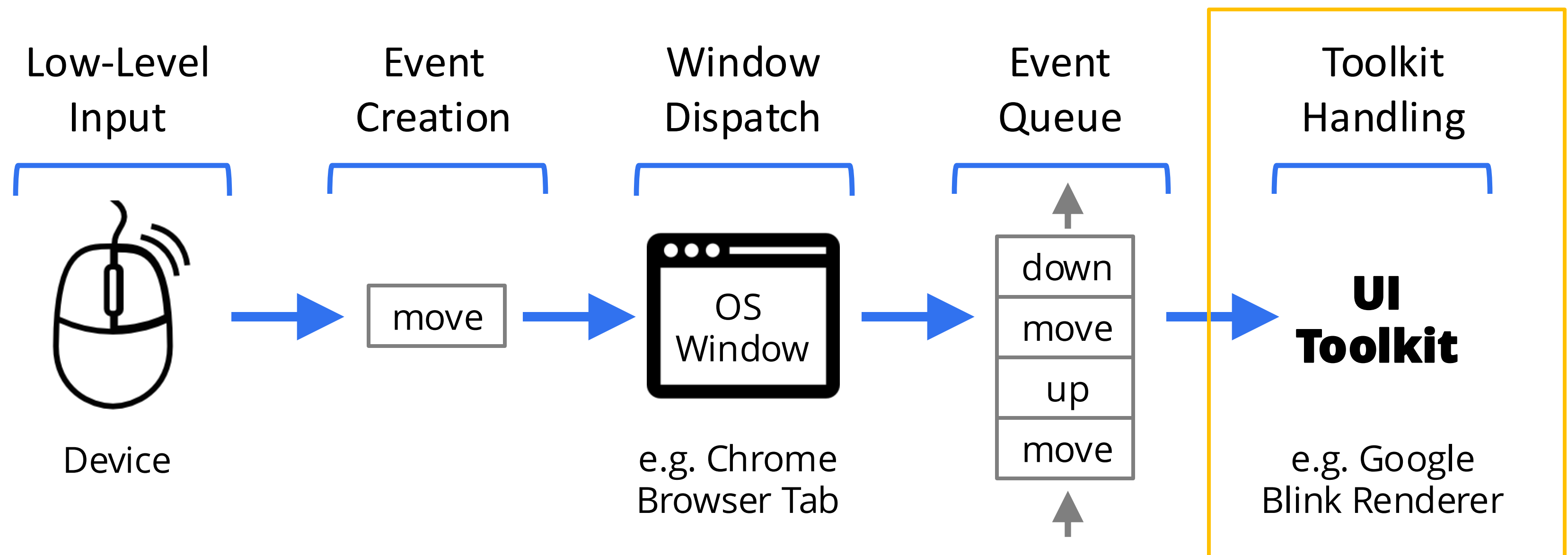
The OS windowing system continually runs an **event loop**

```
loop:
  poll input devices
  create fundamental events
  dispatch fundamental event to focused window
  add fundamental event to window event queue
```



UI Toolkit Run Loop

- The window is running an application which uses a UI Toolkit
  - e.g. DOM Rendering Engine, JavaFX, Cocoa, etc.
- The UI Toolkit handles OS events in its own **run loop**
  - constantly checks for fundamental events in event queue
  - also calls animation timers, re-renders UI, etc.



SimpleKit Windowing System

- SimpleKit simulates how a Windowing System works
  - Uses some HTML DOM events to create *fundamental events*
- **`createWindowingSystem`**
  - (in simplekit windowing-system/ windowing-system.ts)
  - creates a *shared* fundamental event queue

```
const eventQueue: FundamentalEvent[] = [];
```

  - listens to 6 DOM events to use as simulated fundamental events

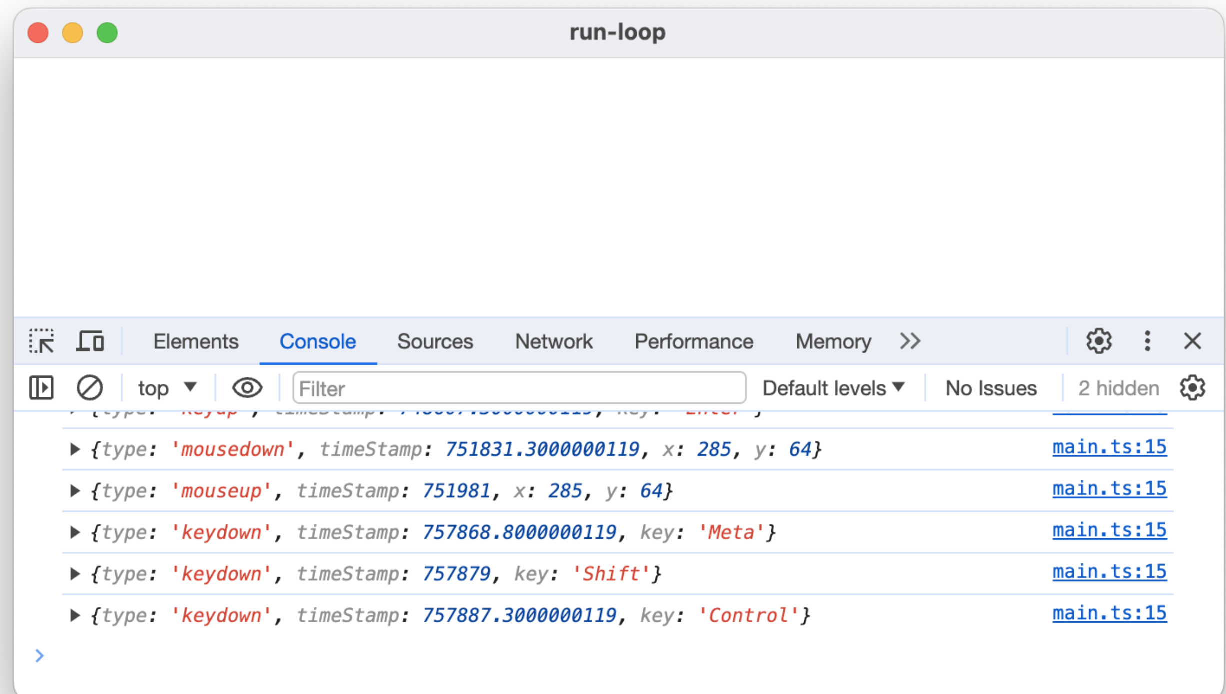
```
window.addEventListener("mousedown", saveEvent);
...
```

  - calls a toolkit **run loop** function at approximately 60 Hz

```
export type RunLoopHandler = (
  eventQueue: FundamentalEvent[],
  time: DOMHighResTimeStamp
) => void;
```

run-loop

- Defines very simple UI run loop
  - the `RunLoopHandler` function
  - log events in queue
- calls `createWindowingSystem`
- What else does a runloop do in a UI toolkit?

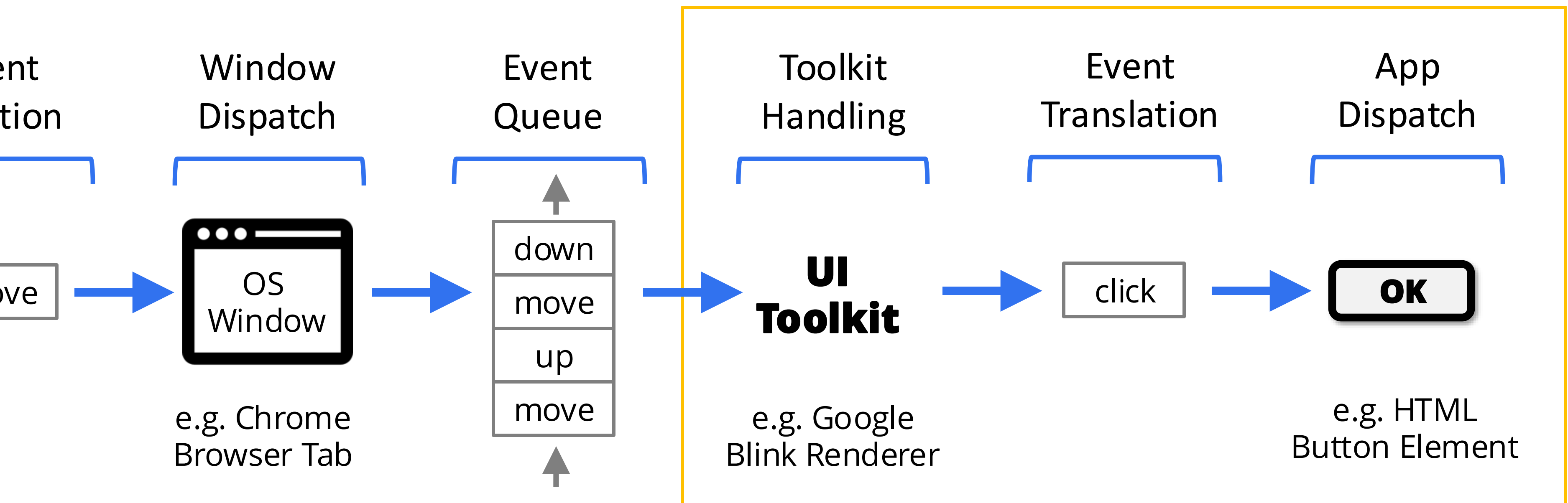


Event Translation in UI Toolkit

The fundamental OS input events are **translated** by the UI Toolkit into UI toolkit events (e.g. "click")

The toolkit events are dispatched to the application

Our event pipeline model is extended as follows:



Event Translation: Higher-level Events

- Common examples of higher-level events:
  - **click**: a mouse button was pressed and released within a certain time window without significant movement
  - **dblclick**: two click events occurred within a small time-window
  - **drag**: the mouse moved while a mouse button is held down
- These can each be modelled as a *state machine*
- The UI Kit also creates events for the fundamental events, like:
  - keydown, keyup, mousedown, mouseup, mousemove, …
- Why is it useful to translate to higher-level events?

SimpleKit Event Classes (in simplekit/events/events.ts)

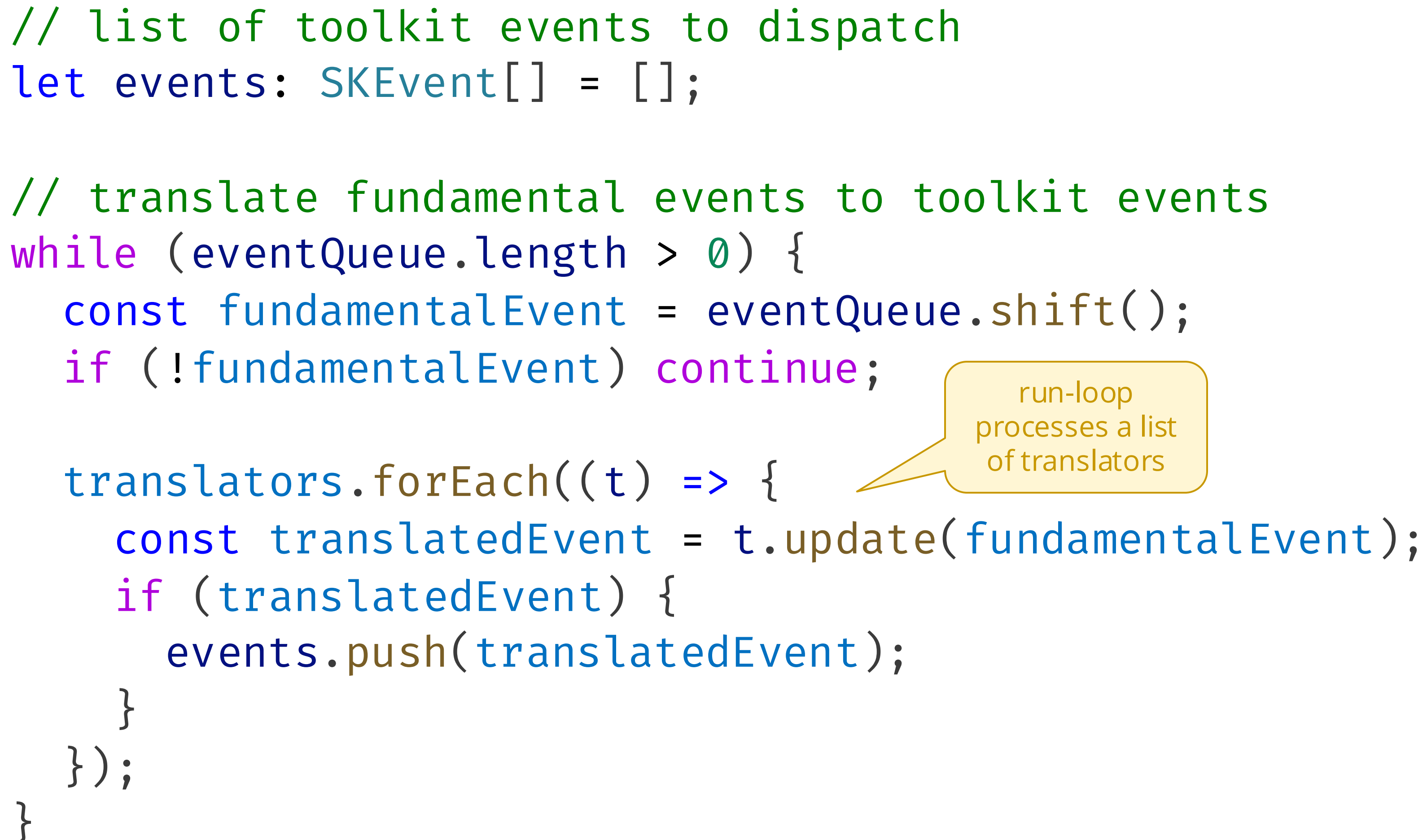
```
export class SKEvent {
  constructor(
    public type: string,
    public timeStamp: number,
    ...
  ) {} }

export class SKMouseEvent extends SKEvent {
  constructor(
    ...
    public x: number,
    public y: number,
    ...
  ) {} }

export class SKKeyboardEvent extends SKEvent {
  constructor(
    ...
    public key: string | null = null,
    ...
  ) {} }
```

showing simplified forms of real classes

translation / run-loop.ts



translation / translators.ts

```
export type EventTranslator = {
  update: (fe: FundamentalEvent) => SKEvent | undefined;
};


export const myTranslator = {
  someStateProperty,
  ...
  update(fe: FundamentalEvent): SKEvent | undefined {
    ...
  },
};
```

most translators need to track state over time

can return specific events inherited from SKEvent, like SKMouseEvent

translators frequently return undefined

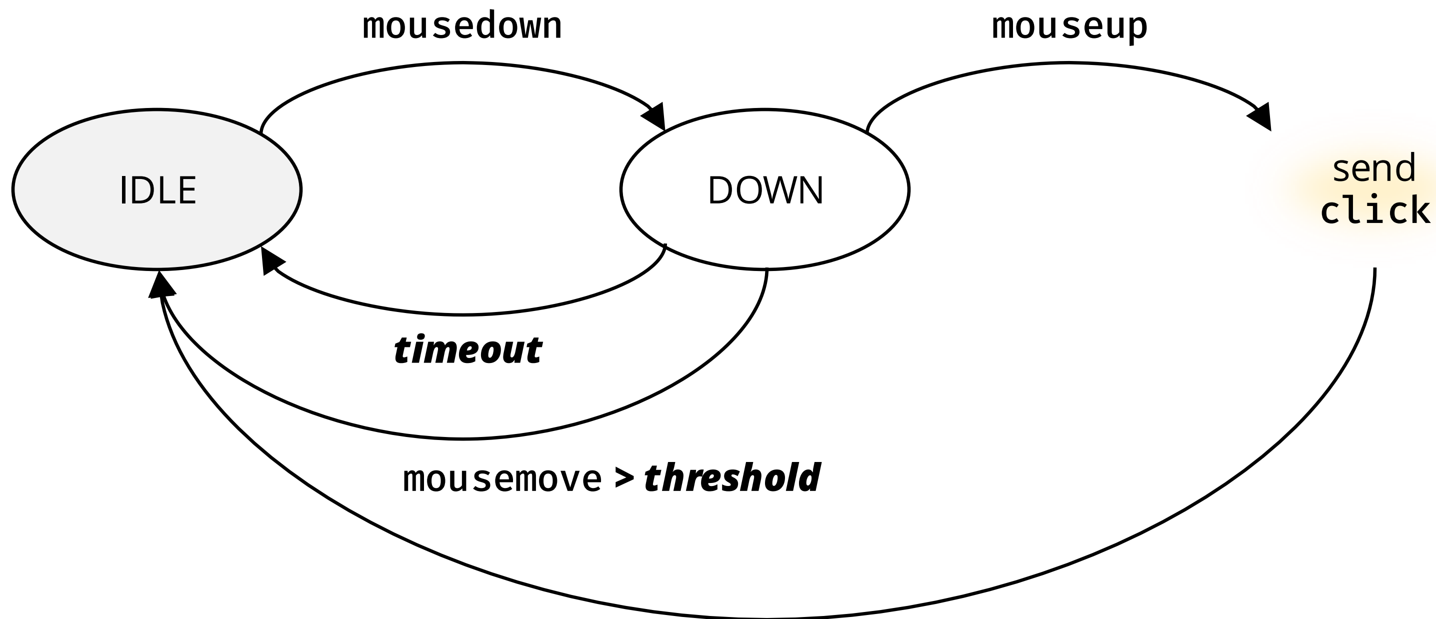
translation / translators.ts

- We need to translate fundamental events to UI toolkit events

```
export const fundamentalTranslator = {
  update(fe: FundamentalEvent): SKEvent {
    switch (fe.type) {
      case "mousedown":
      case "mouseup":
      case "mousemove":
        return new SKMouseEvent(fe.type, fe.timeStamp,
           fe.x || 0, fe.y || 0);
        break;
      case "keydown":
      case "keyup":
        return new SKKeyboardEvent(fe.type, fe.timeStamp,
          fe.key);
        break;
      ...
```

mouse click State Machine

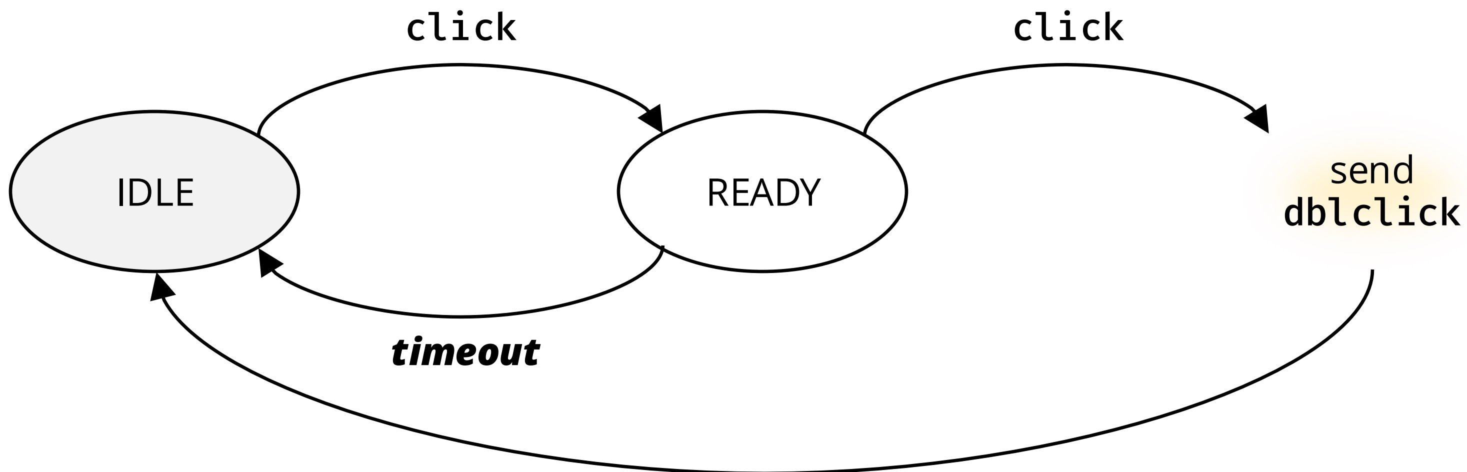
- a mousedown followed by a mouseup within a *short time* and with *little movement* is a mouse button **click**



see code in translators.ts

double click State Machine

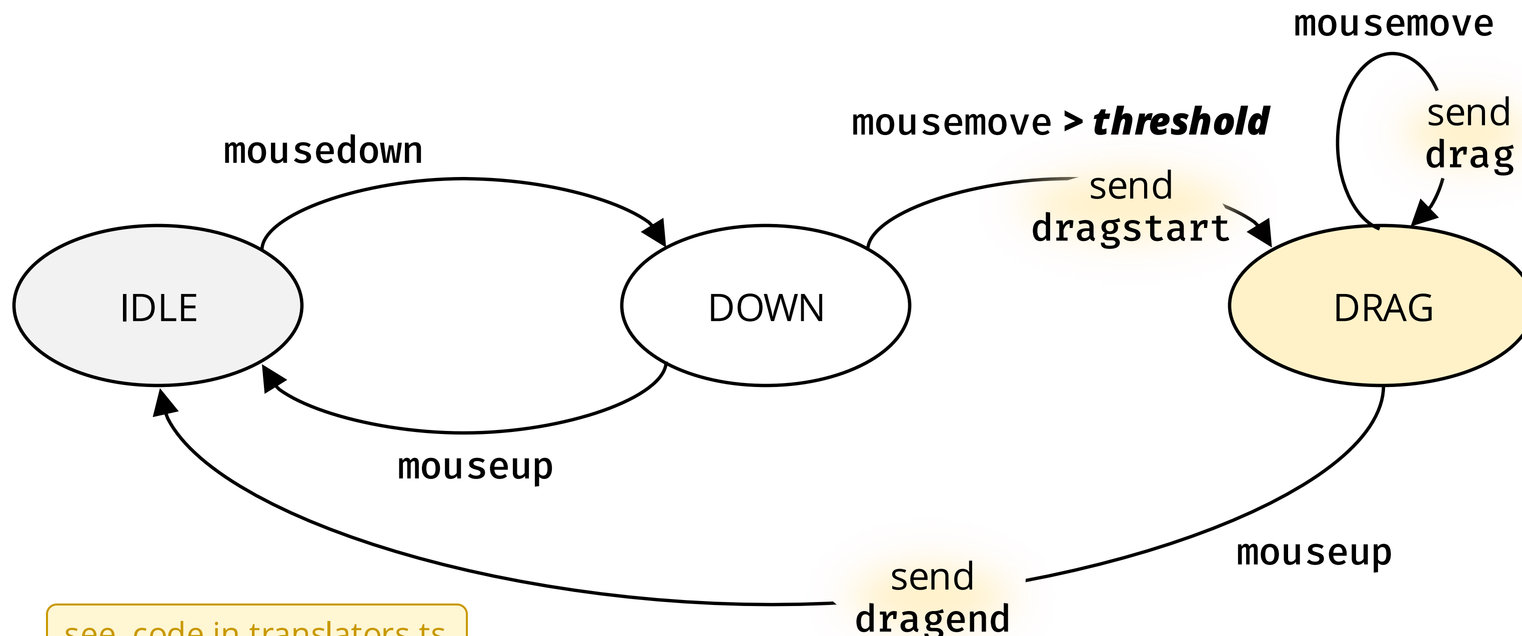
- a click followed by another click with a *short time* is a **dblclick**
- what will happen with single click events?



see code in translators.ts

dragging State Machine

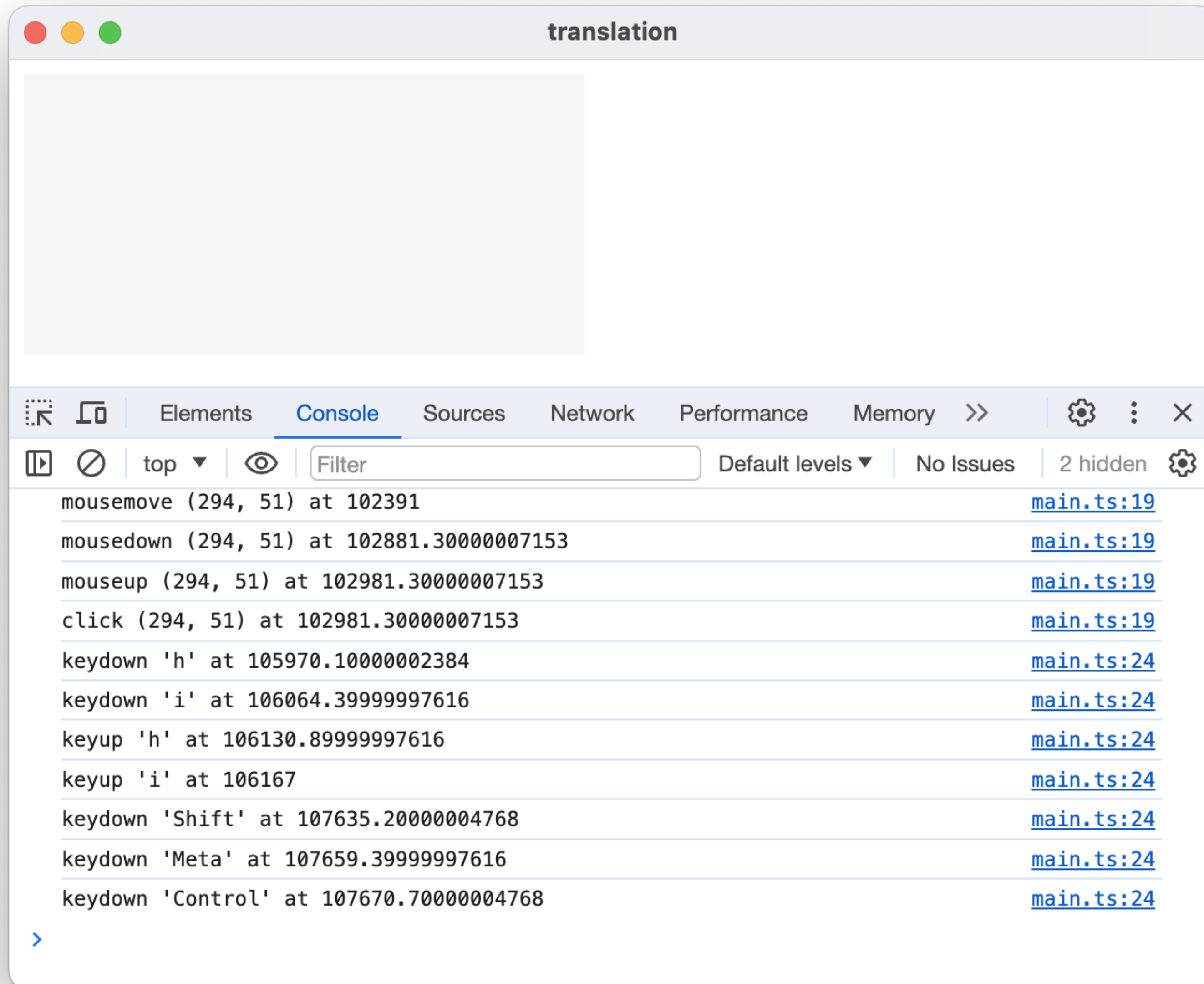
- the dragstart event is sent when the mouse button is held down and the mouse moves *more than a small amount*
- Once in the dragging state, each mousemove triggers a drag event
- mouseup from dragging state also sends a dragend event



UI Toolkit Event Ordering

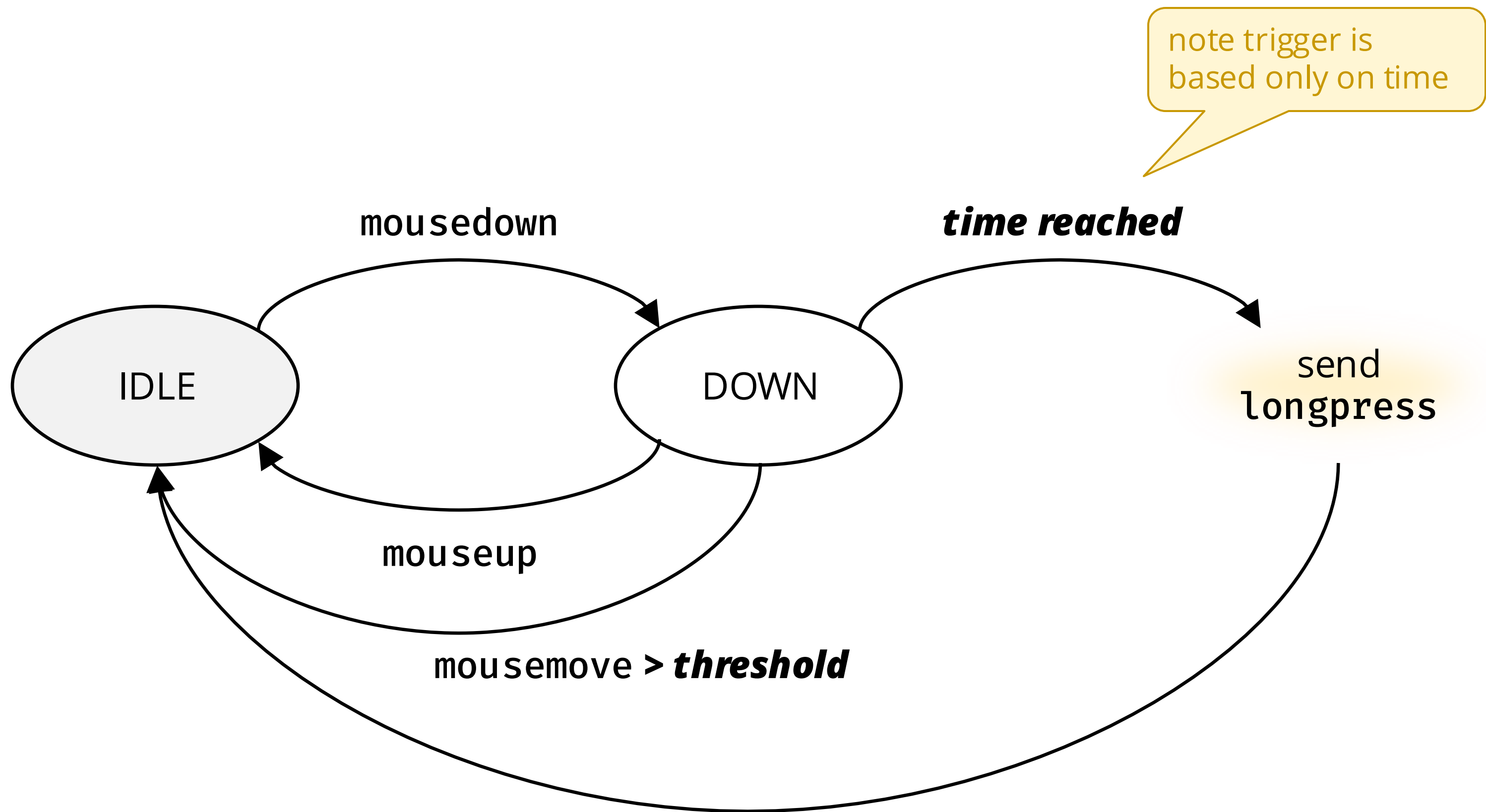
- Translated events will have same timestamp as fundamental event that triggered it
  - *e.g. click will have same timestamp as the mouseup*
- All translated events must be dispatched in deterministic order
  - application can assume sequence if listening to multiple events
    e.g. a click will come after a mouseup

translation / main.ts



longpress State Machine

- a mousedown followed by *little movement* and no mouseup for a *long time* is a **longpress**



simplekit runLoop(…) in canvas-mode.ts

- Some translators need time even when no fundamental events
  - solution is to send a "null" event when no other events
  - if there are events, the translators can use time of those instead

```
if (eventQueue.length == 0) {
  eventQueue.push({
    type: "null",
    timeStamp: time,
  } as FundamentalEvent);
}

// translate fundamental events to toolkit events
while (eventQueue.length > 0) {
  const fundamentalEvent = eventQueue.shift();
  ...
```

Event Translation: Coalesce Frequent Events

- Events like mousedown and mouseup are *discrete state changes*
  - they are not very high frequency (less than 5Hz)
  - each state transition is important
- Events like mousemove describe a *continuing state*
  - they are generated at high frequency (more than 60Hz)
  - the toolkit may not be able to consume them as quickly as they're generated
  - each state transition is typically less important
- Multiple events describing a continuous state can be **coalesced**
  - remove or combine intermediate events since last update
  - avoid coalescing if you want precise movement trajectory without interactive feedback (e.g. saving signature)

see how fast you can click!

https://clickspeedtest.com/

You can see how coalescing works in SimpleKit canvas-mode.ts runLoop(...)

Other UI Events

- A UI Toolkit also dispatches many other events
  - widget receives or loses focus
  - text selection
  - remote data fetching
  - animation starts or ends
  - media events like play, pause, finish
  - clipboard cut, copy or paste
  - socket events
  - worker threads

  ... and many more, see link below

https://developer.mozilla.org/en-US/docs/Web/Events

App Dispatch and Event Binding

- UI Toolkit events need to trigger specific code in the app
- For now, we'll use a simple **dispatch** method:
  - all events are handled in a single app function, e.g.

```
function handleEvent(e: SKEvent) { ... }
```

- For now, we'll also use a simple **binding** method:
  - app code to handle each event is in event handing function, e.g.

```
switch (e.type) {
    case "mousemove":
    // app code here for mousemove
    break;
    case "click":
    // app code here for click
    break;
    ...
```

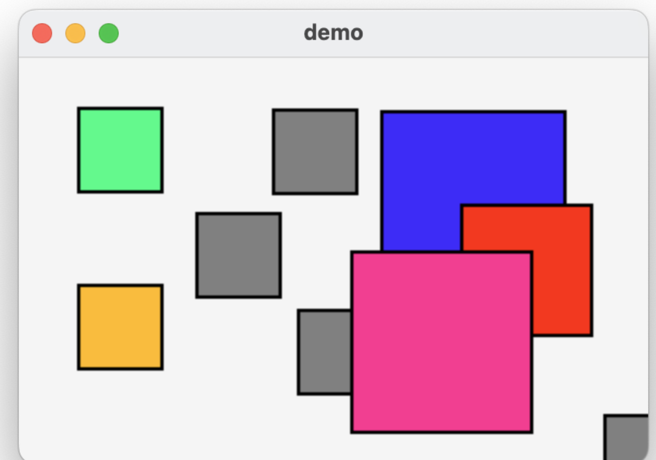
demo

- Example of a more complete SimpleKit canvas-mode app
  - uses `Square` Drawable with DisplayList from drawing lecture
  - handles different events to update state of the square
  - draws squares using a DisplayList
- Event dispatch with "switch statement binding"

  `setSKEventListener`

  `handleEvent`
- Draw with

  `setSKDrawCallback`
- Note **UI state** and **UI drawing code** are separated



Exercise

EXERCISE

1. **Create an app that does the following:**
   - `click` draws a red 50px square
   - `doubleclick` draws a 40px blue square
   - `drag` draws a green 30px square at the ***end of the drag***
   - `keydown` SPACE draws a black 20px square at the mouse position
2. **CHALLENGE 1**
   - `keydown` “x” clears the screen
3. **CHALLENGE 2**:
   - only the last 10 squares are drawn

HINT: use a **DisplayList**