PV252 Seminar 2

Low-level design of UI Elements

How many elements is too many elements?

How many elements is too many elements?

- Only RAM is the limit (in theory).

How many elements is too many elements?

- Only RAM is the limit (in theory).
- In practice, many performance linters already consider 800-1400 elements a problem.

How many elements is too many elements?

- Only RAM is the limit (in theory).
- In practice, many performance linters already consider 800-1400 elements a problem.
- What's going to happen if we use too many elements?

Examples of websites that display a lot of (visually similar) data?

Examples of websites that display a lot of (visually similar) data?

- A few kB of data can generate thousands of UI elements (even small dataset can require a lot of UI).

Examples of websites that display a lot of (visually similar) data?

- A few kB of data can generate thousands of UI elements (even small dataset can require a lot of UI).
- Social media timelines.
- Lists of products (eshops).

Examples of websites that display a lot of (visually similar) data?

- A few kB of data can generate thousands of UI elements (even small dataset can require a lot of UI).
- Social media timelines.
- Lists of products (eshops).
- Photo reels (Google Photos).

Examples of websites that display a lot of (visually similar) data?

- A few kB of data can generate thousands of UI elements (even small dataset can require a lot of UI).
- Social media timelines.
- Lists of products (eshops).
- Photo reels (Google Photos).
- Tables and documents (Google Sheets/Docs/Slides).

Examples of websites that display a lot of (visually similar) data?

- A few kB of data can generate thousands of UI elements (even small dataset can require a lot of UI).
- Social media timelines.
- Lists of products (eshops).
- Photo reels (Google Photos).
- Tables and documents (Google Sheets/Docs/Slides).
- Large folder structures (Github; or file content).

How do you deal with too many UI elements?

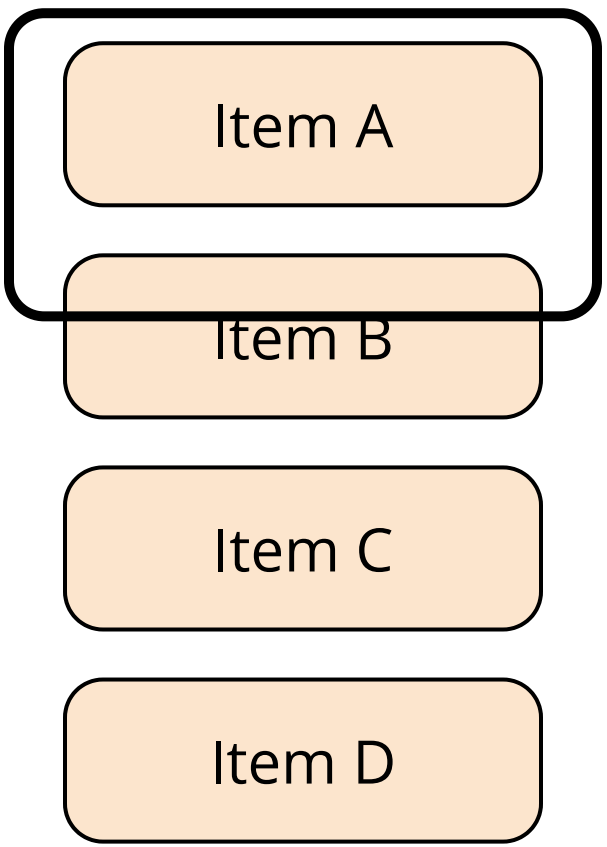
How do you deal with too many UI elements?

- Pagination or hierarchical navigation
  - Works, but can be "immersion breaking" for users.
  - Confusing or impractical for "continuous" data structures, like tables.

How do you deal with too many UI elements?

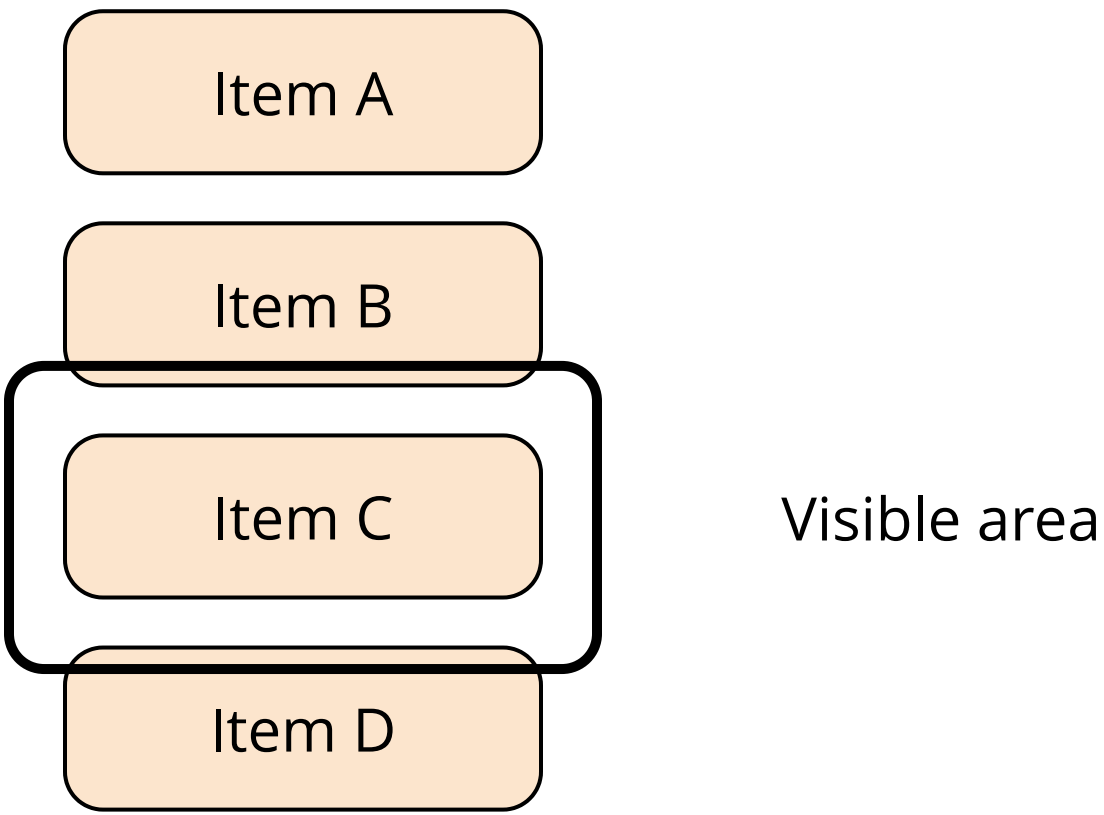
- Pagination or hierarchical navigation
  - Works, but can be "immersion breaking" for users.
  - Confusing or impractical for "continuous" data structures, like tables.
- Lazy List/Grid/Table
  - Also called "windowing", "virtual lists", etc.

"Normal" list

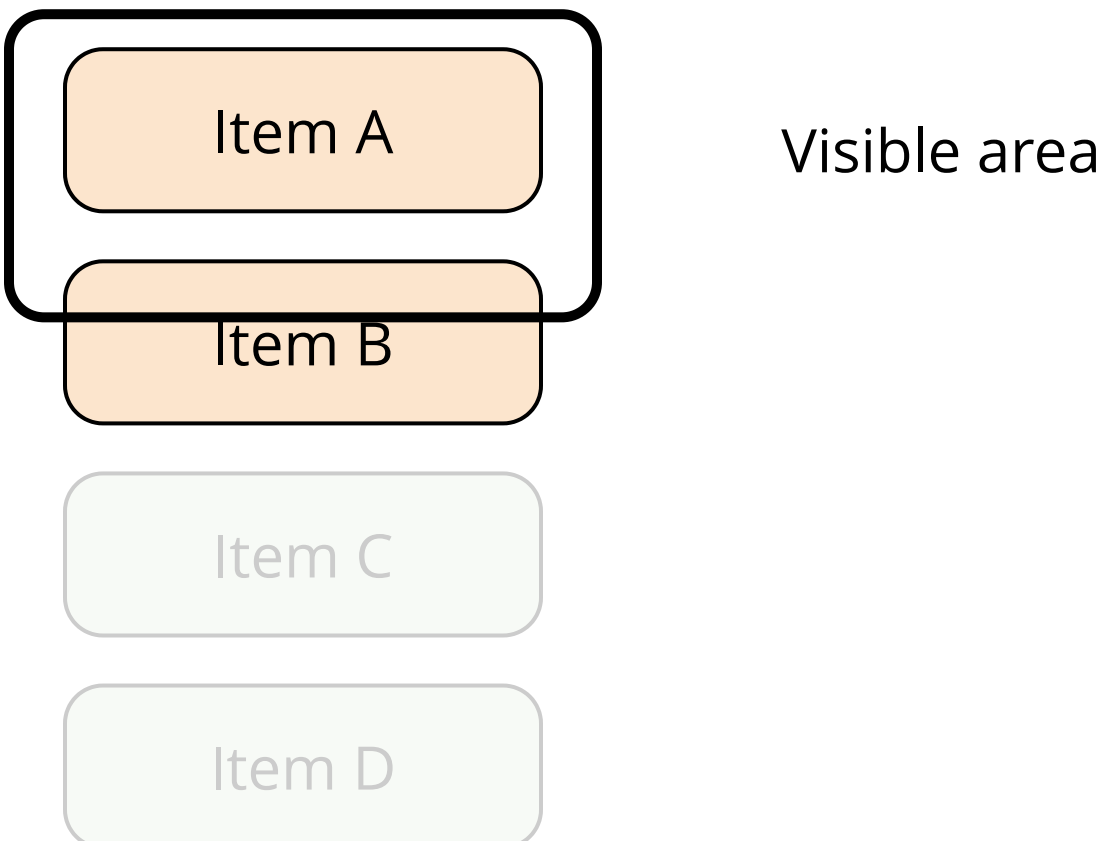


Visible area

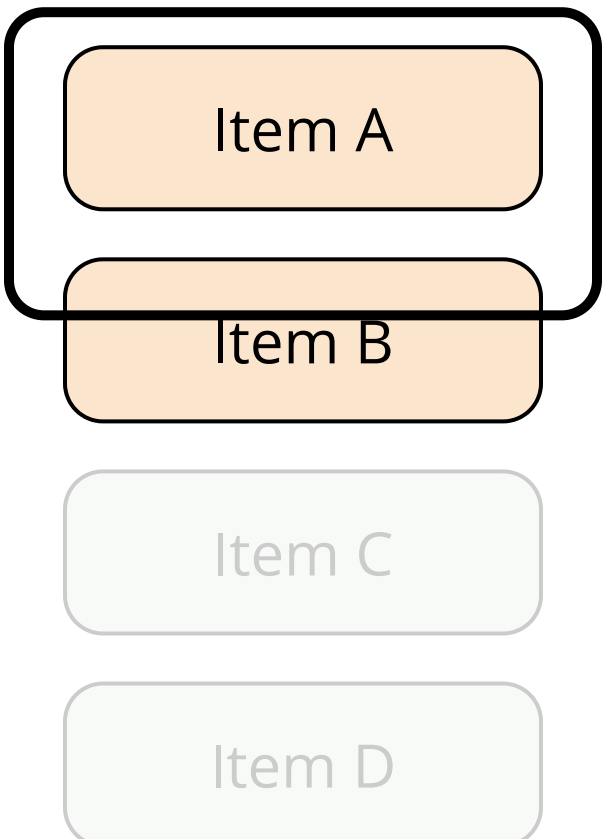
"Normal" list



"Virtual" list



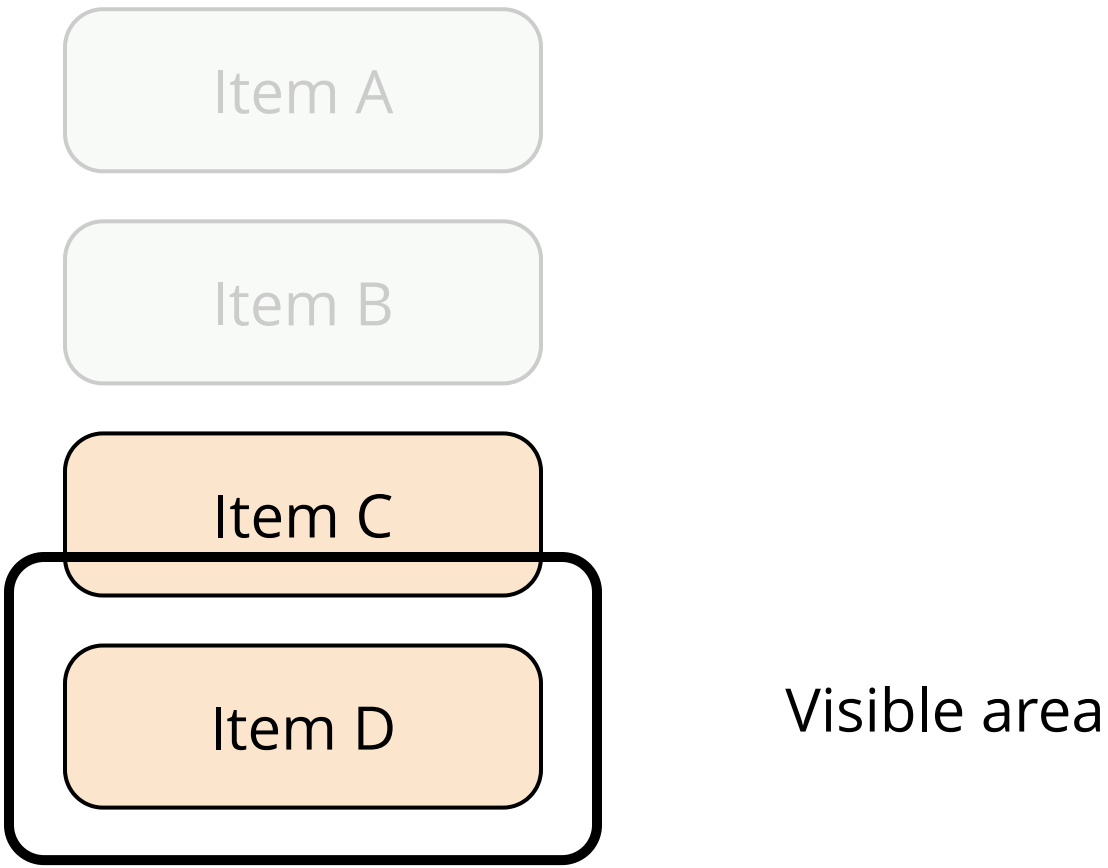
"Virtual" list



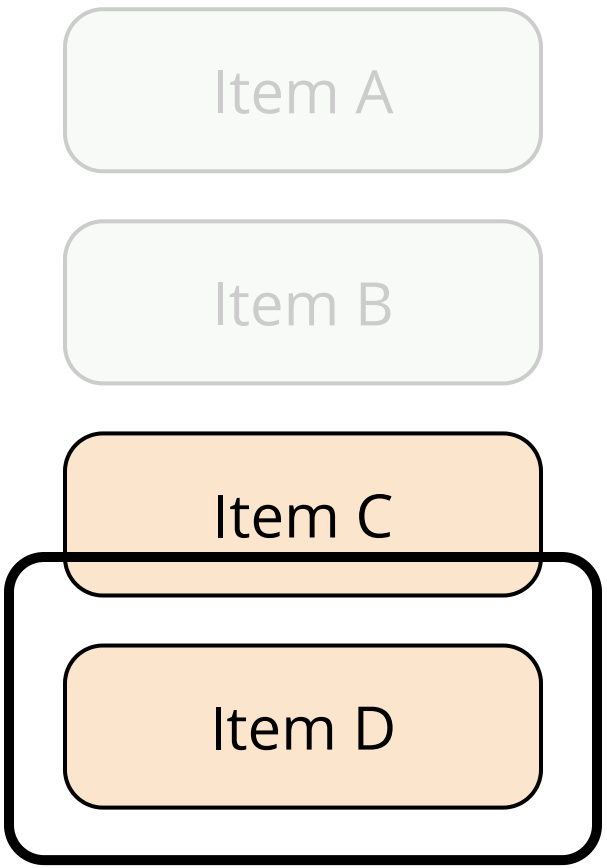
Visible area

Only data needed to display items C and D is kept. The UI elements are not present.

"Virtual" list



"Virtual" list



When visible area moves, the data is used to "render" new UI elements and the unnecessary UI is removed.

Visible area

How do you deal with too many UI elements?

- Pagination or hierarchical navigation
  - Works, but can be "immersion breaking" for users.
  - Confusing for "continuous" data structures, like tables.

How do you deal with too many UI elements?

- Pagination or hierarchical navigation
  - Works, but can be "immersion breaking" for users.
  - Confusing for "continuous" data structures, like tables.
- Lazy List/Grid/Table
  - Also called "windowing", "virtual lists", etc.
  - Harder to implement ("virtual items" usually need to be the same type).
  - Some unexpected behavior with browser features (e.g. ctrl+F search, CSS selectors like ::last-child).

Task for today

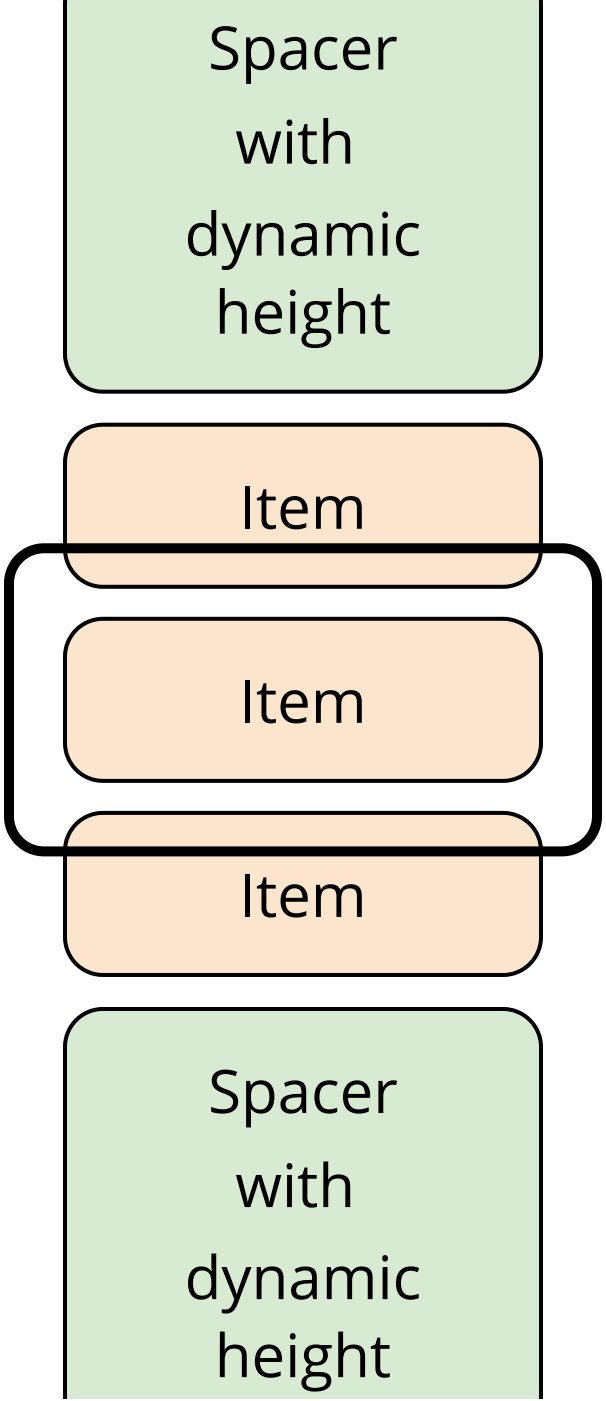
- Implement a "lazy" (virtual) list of famous people from the example as a WebComponent.
- We are not looking for a perfect solution. Assume that:
  - The height of the list component itself is fixed.
  - Items use the same UI hierarchy with (roughly) known height.
  - Elements won't be resized or moved within DOM, list items won't be added or removed on the fly.
  - No need to over-optimize (e.g. if the visible area **meaningfully** changes, you can recompute the HTML for the whole area instead of removing/adding elements one by one, etc.).
  - The user can only interact with the list through "normal" scrolling, not scroll bars or by jumping to a specific item.
  - (if you want to implement any of these extra points, feel free to, but they are not required)

Task for today

- What you need to focus on:
  - Only a fixed number of items is present in the DOM at any moment.
  - The component responds to scroll events by adding/removing list items.
  - List items must not be in the Shadow DOM so that they can be accessed and styled from the outside of the component.
- A "passable" solution can be implemented with <200 lines of code within the provided template. Feel free to make your solution as feature-complete as you want, but the most basic implementation does not need to be overly complicated.

How to get there

- (this is a hint, you don't have to follow it exactly)
- Make sure you know how to build a naive list first.
  - Make the list show rendered items from the given array such that they are in the web component, but not part of the shadow DOM.
- Make sure you can show the first item and manipulate its position based on scroll events.
  - For example, first check that you can make the item stay on screen while the user is scrolling ("simulate" `position: static`).
  - Then try to adjust the position so that the item can scroll, but once it leaves the visible area, it is repositioned to be visible again.
- Still render just one item, but update the content to match the scroll position.
- Add more items to fill the visible area (here, three should be enough, but if you want to dynamically compute the number, you can).



A lot of ways you can "move" the items while scrolling. One straightforward option is to add spacer elements (e.g. an empty div) of fixed height, and then update the height to move up/down items.

Top/bottom margin is another option. Using `position: absolute/relative` with `top` could also work, but special rules for scroll-able areas apply, so it might not be the easiest route.

```
// Get up-to-date scroll position.
el.onscroll = () => { console.log(el.scrollTop) }

// If the UI in a scrollable area changes, the browser
// will automatically try to move to the portion that
// was visible before the UI changed.
const oldScroll = el.scrollTop
... update items inside el ...
el.scrollTop = oldScroll

// Get up-to-date element dimensions (element height
// is not known before it is first rendered inside the DOM).
const observer = new ResizeObserver(() => {
    console.log(el.offsetHeight)
})
observer.observe(el)


// How do I insert items into <slot>?
<my-fancy-list>
  <div class=list> // shadow DOM
    <slot></slot>
  </div>
</my-fancy-list>

const list = ... // get reference to my-fancy-list instance
// "Normal" manipulation of child elements will not affect the
// shadow DOM, but will add/remove elements from <slot> directly.
list.innerHTML = "<span>Some content in the list.</span>"
list.appendChild(someElement)
```

```
1 git checkout -b list-component --track upstream/list-component
```

(if you have the repository forked on
github from previous seminar)