Lab 8: Animation

SUMMARY

In this lab, we will learn:

- What different methods do we have to transition and animate elements and when to use each?
- What considerations should we take into account for using animations effectively?

▼ TABLE OF CONTENTS

Submission

To get checked off for the lab, please record a 2 minute video with the following components:

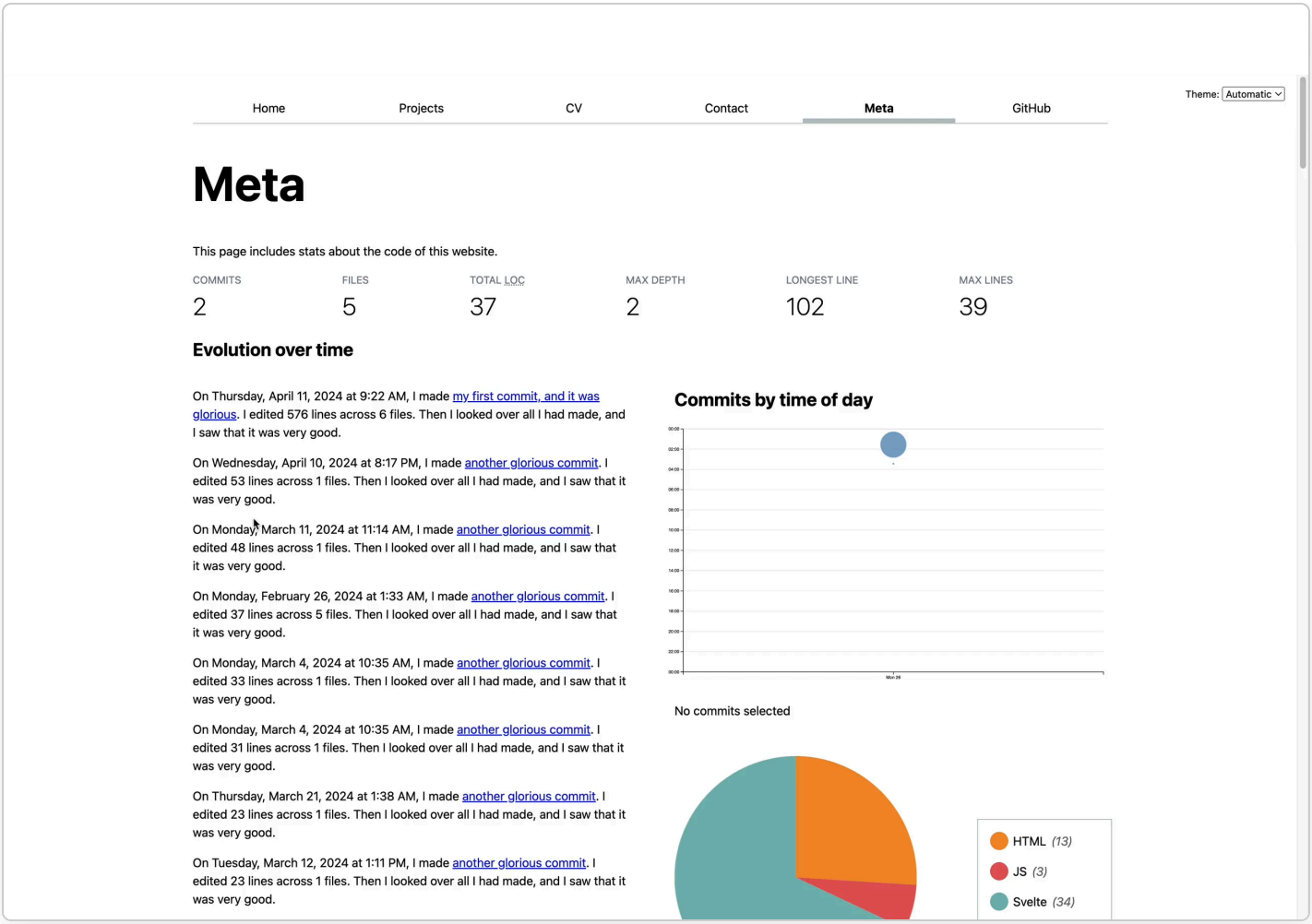
1. Present your interactive narrative visualization
2. Show you interacting with your visualization.
3. Share the most interesting thing you learned from this lab.

Videos longer than 2 minutes will be trimmed to 2 minutes before we grade, so make sure your video is 2 minutes or less.

Slides

What will we make?

In this lab, we will go back to the Meta page of our portfolio, and convert it to an interactive narrative visualization that shows the progress of our codebase over time (you may **ignore the pie chart part** in the demo since we did not explicitly implement it for our meta tab).



Step 0: Clean Up

To make your code structure a little nicer, we will first complete the following refactoring steps.

> **FILE(S) EDITED IN THIS SECTION**
>
> `src/meta/main.js`

Currently, our `selectedCommits` variable is meant to reactively update and depends on `brushSelection`:

```
selectedCommits = brushSelection ? commits.filter(isCommitSelected) : [];
```

We also have an `isCommitSelected()` function, which checks of a commit is within the `brushSelection` and looks like this:

```
function isCommitSelected(commit) {
  if (!brushSelection) {
    if (manualSelection) {
```

```
      return manualSelection.has(commit);
    }

    return false;
  }

  let min = { x: brushSelection[0][0], y: brushSelection[0][1] };
  let max = { x: brushSelection[1][0], y: brushSelection[1][1] };
  let x = xScale(commit.date);
  let y = yScale(commit.hourFrac);

  return x >= min.x && x <= max.x && y >= min.y && y <= max.y;
}
```

However, `brushSelection` is actually only updated in one place: the `brushed()` function. We don't really need to keep it around once we've converted it to selected commits. Let's update the `brushed()` function to update `selectedCommits` directly:

```
function brushed(evt) {
  let brushSelection = evt.selection;
  selectedCommits = !brushSelection
    ? []
    : commits.filter((commit) => {
        let min = { x: brushSelection[0][0], y: brushSelection[0][1] };
        let max = { x: brushSelection[1][0], y: brushSelection[1][1] };
        let x = xScale(commit.date);
        let y = yScale(commit.hourFrac);

        return x >= min.x && x <= max.x && y >= min.y && y <= max.y;
      });
}
```

Then `isCommitSelected()` can be much simpler:

```
function isCommitSelected(commit) {
  return selectedCommits.includes(commit);
}
```

And `selectedCommits` becomes a variable that you may declare at the top level of your code:

```
let selectedCommits = [];
```

Also, we can now make the colors of individual commits on mouse events consistent with brush selections. We can add this line of code to event handling of `mouseenter` and `mouseleave` (you should have a section that handles these two events in your **scatter plot** code):

```
d3.select(event.currentTarget).classed('selected', ...); // give it a corresponding boolean value
```

Step 1: Evolution visualization

> **FILE(S) EDITED IN THIS SECTION**
>
> `src/meta/main.js` and `src/meta/index.html`

In this step, we will create an interactive timeline visualization that shows the evolution of our repo by allowing us to move a slider to change the date range of the commits we are looking at.

Step 1.1: Creating the filtering UI

In this step we will create a slider, bind its value to a variable, and display the date and time it corresponds to. It's very familiar to what we did in the previous lab.

First, let's create a new variable, `commitProgress`, that will represent the maximum time we want to show as a percentage of the total time:

```
let commitProgress = 100;
```

To map this percentage to a date, we will need a new time scale. Once we have our scale, we can easily get from the 0-100 number to a date:

```
let timeScale = d3.scaleTime([d3.min(commits, d => d.datetime), d3.max(commits, d => d.datetime)], [0, 100
let commitMaxTime = timeScale.invert(commitProgress);
```

We are now ready to add our filtering UI in `index.html`. This is largely a repeat of what we did in Lab 7:

- A slider (`<input type=range>`) with a min of 0 and max of 100 and bind the slider value to `commitProgress`.
- A `<time>` element to display the commit time using `commitMaxTime.toLocaleString()`.
- A `<label>` *around* the slider and `<time>` element with some explanatory text (e.g. "Show commits until:").

Where you put it on the page is up to you. I placed it on top of my scatter plot (the `<div>` element with id `chart`). I wrapped the `<label>` inside of `<div>`, to which I applied a `flex: 1` and `align-items: baseline` to align them horizontally. Then I gave `<time>` a `margin-left: auto` to push it all the way to the right. **Note you should apply these style rules as specifically as possible (use ID selectors)!**

> **TIP**
> Feel free to use any settings you like. In the screencasts below, I used `dateStyle: "long"` and `timeStyle: "short"`. You may pass these options into `toLocaleString()` method as an object.

If everything went well, your slider should now be working!

> **NOTE**
> To make the time string present, you should have the following lines:
>
> ```
> const selectedTime = d3.select('#selectedTime');
> selectedTime.textContent = timeScale.invert(commitProgress).toLocaleString();
> ```
>
> Figure out where's the most appropriate position to place these.

Step 1.2: Filtering by `commitMaxTime`

Let's now ceate a new `filteredCommits` variable that will reactively filter `commits` by comparing `commit.datetime` with `commitMaxTime` and only keep those that are **less than** `commitMaxTime`.

We can now replace `commits` with `filteredCommits` in several places (these varies depending on the exact implementation you did for Lab 6 so please be mindful):

- The `xScale` domain for commit time
- The `rScale` domain for each scatter's radius

- The `brushed()` function that updates the `selectedCommits` variable
- Your summary stats

Just to demonstrate, let's take the `createScatterplot()` method that you may have created:

```
function createScatterplot() {
    // you may have wrote the following lines
    const width = 1000;
    const height = 600;
    const svg = d3.select('#chart').append('svg')...
    xScale = d3.scaleTime()...
    yScale = d3.scaleLinear()...
}
```

You should update it to the following:

```
function updateScatterplot(filteredCommits) {
  // same as before

  d3.select('svg').remove(); // first clear the svg
  const svg = d3.select('#chart').append('svg')...

  xScale = d3.scaleTime().domain(d3.extent(filteredCommits, (d) => d.datetime))...

  /// same as before

  svg.selectAll('g').remove(); // clear the scatters in order to re-draw the filtered ones
  const dots = svg.append('g').attr('class', 'dots');

  // same as before

  const [minLines, maxLines] = d3.extent(filteredCommits, (d) => d.totalLines);
  const rScale = d3.scaleSqrt().domain([minLines, maxLines])...;

  // same as before

  dots.selectAll('circle').remove();
  dots.selectAll('circle').data(filteredCommits).join('circle')...
```

```
  // same as before
}
```

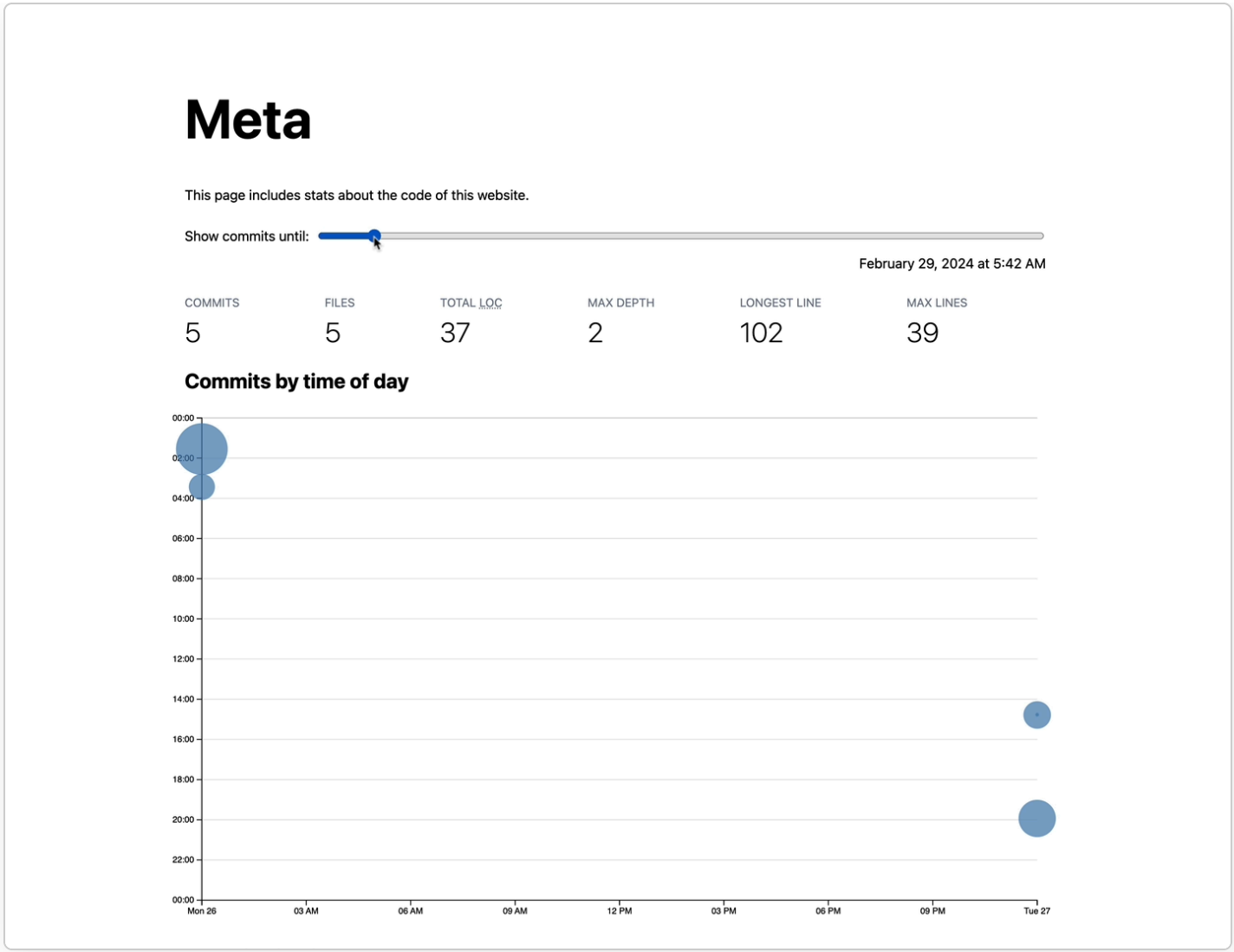
Then, you can replace your previous `createScatterplot()` function with `updateScatterplot(commits)` to retain the scattor plot of commits on page load up. Next you can update the scatter plot by calling `updateScatterplot(filteredCommits)` once you finish filtering `commits` by `commitMaxTime`.

You may consider adding this function call to the `updateTimeDisplay()` method:

```
function updateTimeDisplay() {
  commitProgress = Number(timeSlider.value);
  // what ever you have previously
  ...
  filterCommitsByTime(); // filters by time and assign to some top-level variable.
  updateScatterplot(filteredCommits);
}
```

Step 1.3: Entry transitions with CSS

Notice that even though we are now getting a nice transition when an existing commit changes radius, there is no transition when a new commit appears.

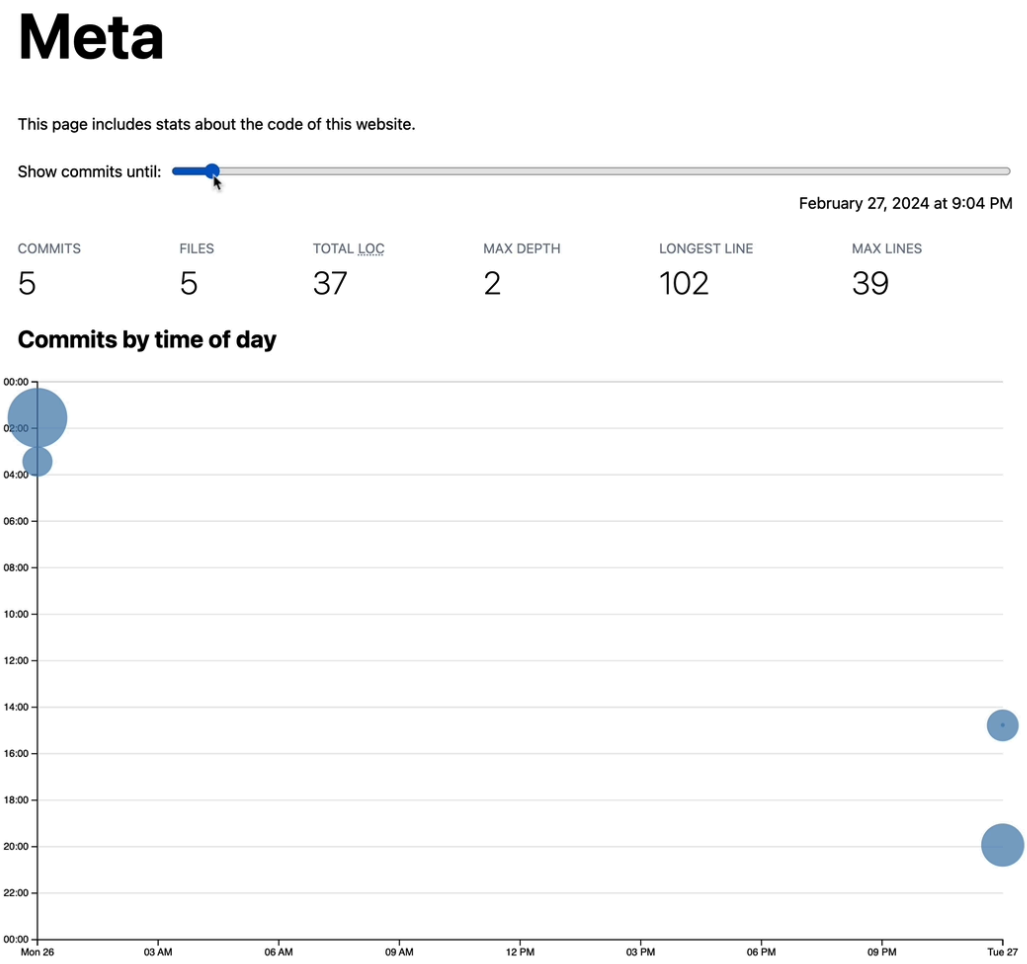


This is because CSS transitions fire for state changes where both the start and end changes are described by CSS. A new element being added does not have a start state, so it doesn't transition. We can use CSS transitions to help resolve this, by explicitly telling the browser what the start state should be. That's what the `@starting-style` rule is for!

Inside the `circle` CSS rule, add a `@starting-style` rule:

```
@starting-style {
  r: 0;
}
```

If you preview again, you should notice that that's all it took, new circles are now being animated as well!



GOING FURTHER

You might notice that the largest circles and the smallest circles are *both* transitioning with the same *duration*, which means dramatically different *speeds*. We may decide that this is desirable: it means all circles are appearing at once. However, if you want to instead keep speed constant, you can set an `--r` CSS variable on each `circle` element with its radius, and then set the transition duration to e.g. `calc(var(--r) / 100ms)`. You can do that only for `r` transitions like so:

```
transition: all 200ms, r calc(var(--r) * 100ms);
```

Step 2: The race for the biggest file!

In this step we will create a unit visualization that shows the relative size of each file in the codebase in lines of code, as well as the type and age of each line.

Step 2.1: Adding unit visualization for files

> **FILE(S) EDITED IN THIS SECTION**
>
> `src/meta/main.js` and `src/style.css`

We want to display the file details for the commits we filtered. We want this section to go after the scatter plot, but for now let's add it right after our filtering slider as that makes development faster.

First, let's obtain the file names and lines associated with each file.

```
let lines = filteredCommits.flatMap((d) => d.lines);
let files = [];
files = d3
  .groups(lines, (d) => d.file)
  .map(([name, lines]) => {
    return { name, lines };
  });
```

Now that we have our files, let's output them (filenames and number of lines). We will use a `<dl>` element (but feel free to make different choices, there are many structures that would be appropriate here) to give it a simple structure.

```
<dl class="files">
  <!-- we want the following structure for each file-->
  <div>
    <dt>
      <code>{file.name}</code>
    </dt>
    <dd>{file.lines.length} lines</dd>
  </div>
  <div>
    ...
  </div>
</dl>
```

It should be clear by now what we need from D3 to achieve this:

```
d3.select('.files').selectAll('div').remove(); // don't forget to clear everything first so we can re-renc
let filesContainer = d3.select('.files').selectAll('div').data(files).enter().append('div');

filesContainer.append('dt').append('code').text(d => ...); // TODO
filesContainer.append('dd').text(d => ...); // TODO
```

We should style the `<dl>` as a grid so that the filenames and line counts are aligned. The only thing that is a bit different now is that we have a `<div>` around each `<dt>` and `<dd>`. To prevent that from interfering with the grid we should use Subgrid:

```
.files > div {
  grid-column: 1 / -1;
  display: grid;
  grid-template-columns: subgrid;
}
```

Then we can just apply `grid-column: 1` to the `<dt>`s and `grid-column: 2` to the `<dd>` as usual.

At this point, our “visualization” is rather spartan, but if you move the slider, you should already see the number of lines changing!

src/routes/meta/CommitScatterplot.svelte	252 lines
src/lib/Pie.svelte	239 lines
src/routes/meta/+page.svelte	182 lines
src/lib/Scroller.svelte	98 lines
src/routes/meta/FileLines.svelte	93 lines
src/routes/+page.svelte	58 lines
src/routes/projects/+page.svelte	54 lines
src/routes/+layout.svelte	44 lines
src/lib/Project.svelte	35 lines
src/lib/Projects.svelte	22 lines
src/lib/transition.js	20 lines
src/app.html	15 lines
src/routes/contact/+page.svelte	10 lines
src/lib/index.js	1 lines

> **NOTE**
> You may see different summary stat changes depending on which you implemented from Lab 6. And if you are having trouble aligning things, revisit Lab 2 Step 4.3, where we first used sub-grids to align your contact form.

Step 2.2: Making it look like an actual unit visualization

For a unit visualization, we want to draw an element per data point (in this case, per line committed), so let's do that. All we need to do is replace the contents of the `<dd>` element with more `<div>`, each corresponding to one line:

```
<!-- we want to achieve this -->
<dd>
  <div class="line"></div>
</dd>
```

To do so, simply add to where we were appending `<dd>` using D3 selections previous:

```
// TODO, append divs and set each's class attribute
filesContainer.append('dd')
```

```
    .selectAll('div')
    .data(d => d.lines)
    ...
```

> **TIP**
> Seeing the total number of lines per file is still useful, so you may want to add it in the `<dt>`. I used a `<small>` element, gave it `display: block` so that it's on its own line, and styled it smaller and less opaque. You can set both `<code>` and `<small>` tags' contents using `.html()` method. You can revisit it in Lab 5 Step 2.2

And then add some CSS to make it look like a unit visualization:

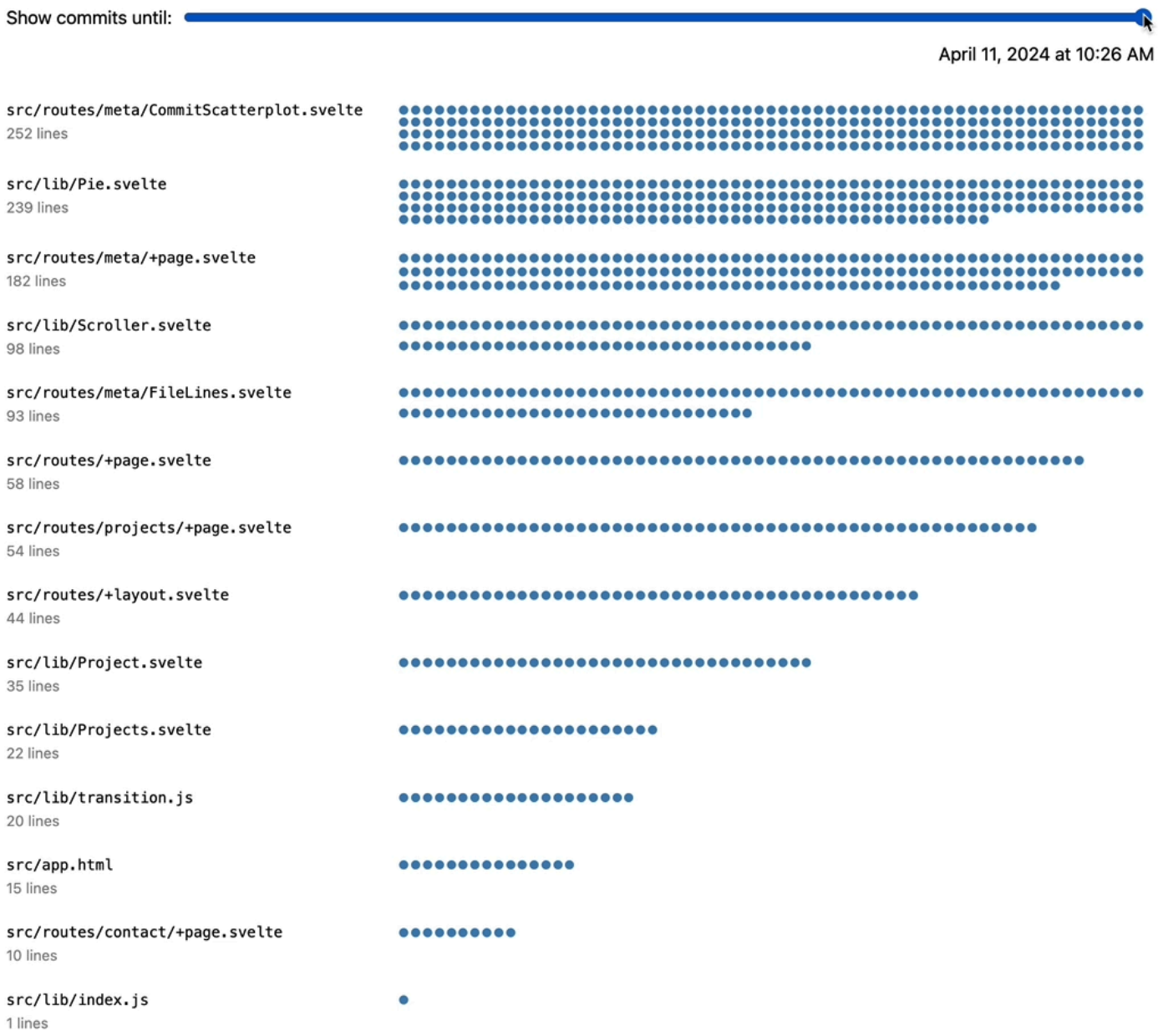
```
.line {
  display: flex;
  width: 0.5em;
  aspect-ratio: 1;
  background: steelblue;
  border-radius: 50%;
}
```

Last, we want to make sure these dots wrap and are tightly packed, so we need to add some CSS for the `<dd>` elements to allow this:

```
dd {
  grid-column: 2;
  display: flex;
  flex-wrap: wrap;
  align-items: start;
  align-content: start;
  gap: 0.15em;
  padding-top: 0.6em;
  margin-left: 0;
}
```

At this point, we should have an actual unit visualization!

It should look something like this:



Step 2.3: Sorting files by number of lines

Our visualization is not really much of a race right now, since the order of files seems random. We need to sort the files by the number of lines they contain in descending order. We can do that in the same reactive block where we calculate `files`:

```
files = d3.sort(files, (d) => -d.lines.length);
```

Step 2.4: Varying the color of the dots by technology

> **CAVEAT**

> Let's first remove the CSS rule on `.line`'s background in order for us to render different lines of code with distinctive colors.

Our visualization shows us the size of the files, but not all files are created equal. We can use color to differentiate the lines withn each file by technology.

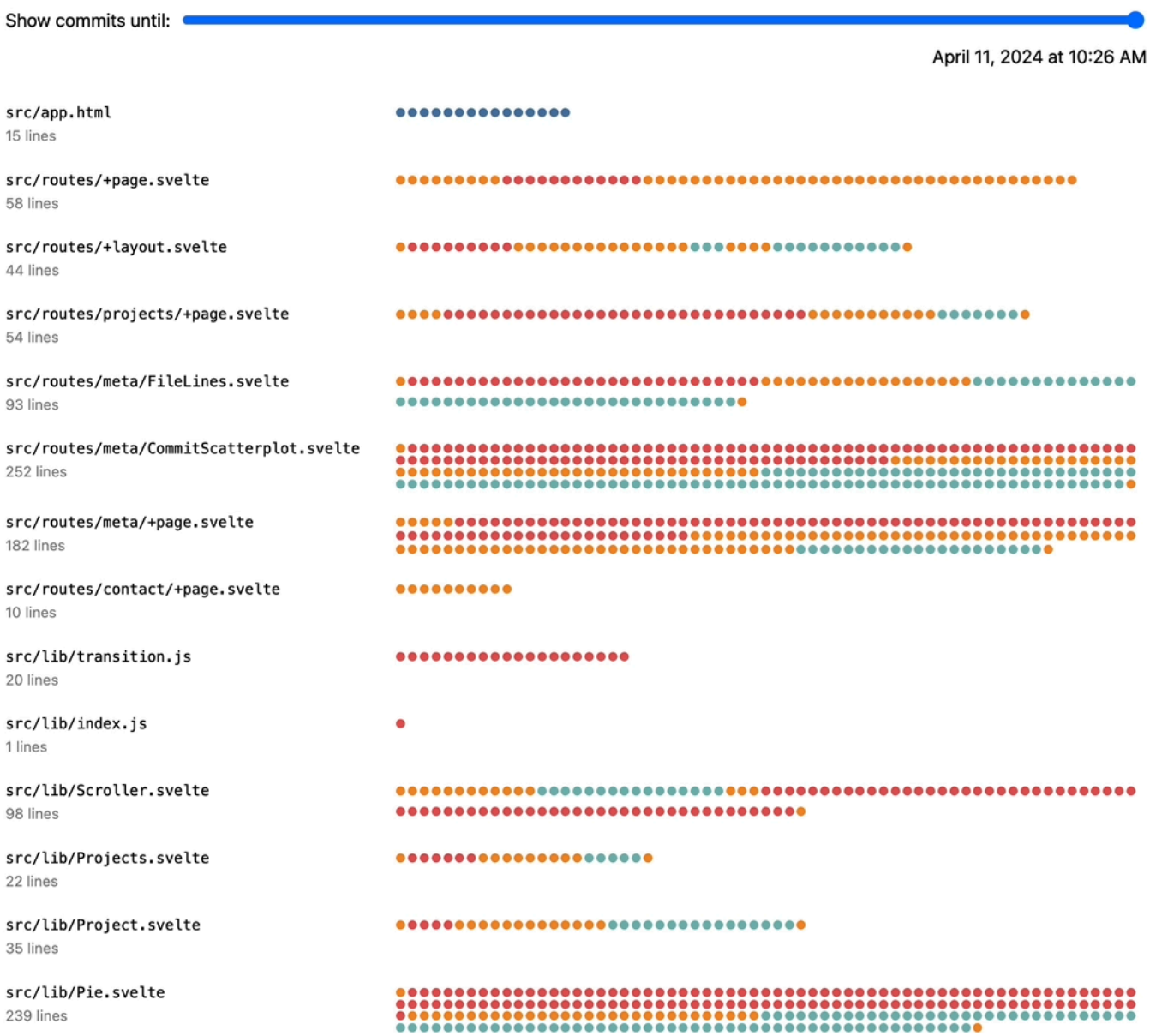
Let's create an ordinal scale that maps technology ids to colors:

```
let fileTypeColors = d3.scaleOrdinal(d3.schemeTableau10);
```

Then, we can use this scale to color the dots:

```
filesContainer.append('dd')
              .selectAll('div')
              .data(d => d.lines)
              ... // same as before
              .style('background', ...); // TODO, apply the color scale based on line type
```

Much better now!



Step 3: Scrollytelling Part 1 (commits over time)

> **FILE(S) EDITED IN THIS SECTION**
>
> `src/meta/main.js`, `src/style.css`, and `src/meta/index.html`.

So far, we have been progressing through these visualizations by moving a slider. However, these visualizations both tell a story, the story of how our repo evolved. Wouldn't it be cool if we could *actually* tell that story in our own words, and have the viewer progress through the visualizations as they progress through the narrative?

Let's do that!

Step 3.0: Making our page a bit wider, if there is space

Because our scrolly will involve a story next to a visualization, we want to be able to use up more space, at least in large viewports.

We can do this only for the Meta page, by adding a CSS rule where the selector is `:global(body)`.

Then, within the rule, we want to set the `max-width` to `120ch` (instead of `100ch`), but only as long as that doesn't exceed 80% of the viewport width. We can do that like this:

```
max-width: min(120ch, 80vw);
```

Step 3.1: Implementing a Scrolly

Let's implement our own Scrolly. We will first start by adding a new `<div>`-level addition to our HTML file and introducing a few more top-level variables that will help us define the layout of our scrolling window.

First, let's restructure how we want to display the scatterplot along with our scrollytelling window in `src/meta/index.html`. Put the following in place of where you originally had `<div id='chart'></div>`:

```
<div id="scrollytelling">
    <div id="scroll-container">
        <div id="spacer"></div>
        <div id="items-container"></div>
    </div>
    <!-- our old scatterplot div -->
    <div id="chart"></div>
</div>
```

Let's add some styling to it too (you are free to improvise any of the following style rules):

```
#scrollytelling {
  grid-column: 1 / -1;
  display: grid;
  grid-template-columns: subgrid;
}

/* feel free to play with this to make your scrolly more seemless with your plot */
#scroll-container {
```

```
  grid-column: 1;
  position: relative;
  width: 95%;
  height: 350px;
  overflow-y: scroll;
  border: 1px solid #ccc;
  margin-bottom: 50px;
}

#chart {
  grid-column: 2;
}

#spacer {
  position: absolute;
  top: 0;
  left: 0;
  width: 100%;
  background: none; /* transparent */
  pointer-events: none;
}

#items-container {
  position: absolute;
  top: 0;
  left: 0;
  width: 100%;
}

.item {
  height: 30px;
  padding: 10px;
  box-sizing: border-box;
  border-bottom: 2px solid #eee;
}
```

Then let's introduce the global variables:

```
let NUM_ITEMS = 100; // Ideally, let this value be the length of your commit history
let ITEM_HEIGHT = 30; // Feel free to change
let VISIBLE_COUNT = 10; // Feel free to change as well
let totalHeight = (NUM_ITEMS - 1) * ITEM_HEIGHT;
const scrollContainer = d3.select('#scroll-container');
const spacer = d3.select('#spacer');
spacer.style('height', `${totalHeight}px`);
const itemsContainer = d3.select('#items-container');
scrollContainer.on('scroll', () => {
  const scrollTop = scrollContainer.property('scrollTop');
  let startIndex = Math.floor(scrollTop / ITEM_HEIGHT);
  startIndex = Math.max(0, Math.min(startIndex, commits.length - VISIBLE_COUNT));
  renderItems(startIndex);
});
```

Now as you may already see, it's time to implement the `renderItems()` method to adaptively show us commits info as we scroll through it. It's logic is actually fairly straightforward. Upon function call, first erase all previous scrolling results like we always do, then take a new slice of commits and bind it to the commit item container. Here is how it works:

```
function renderItems(startIndex) {
    // Clear things off
    itemsContainer.selectAll('div').remove();
    const endIndex = Math.min(startIndex + VISIBLE_COUNT, commits.length);
    let newCommitSlice = commits.slice(startIndex, endIndex);
    // TODO: how should we update the scatterplot (hint: it's just one function call)
    ...
    // Re-bind the commit data to the container and represent each using a div
    itemsContainer.selectAll('div')
                  .data(newCommitSlice)
                  .enter()
                  .append('div')
                  ... // TODO: what should we include here? (read the next step)
                  .style('position', 'absolute')
                  .style('top', (_, idx) => `${idx * ITEM_HEIGHT}px`)
}
```

Step 4.2: Creating a dummy narrative

As you may have guessed, the whole point of implementing a scrolly is to represent something meaningful for the narrative. Don't spend long on it; you can even generate it with ChatGPT as long as you check that the result is coherent, relevant, and tells a story that complements to the visualization next to it without simply repeating information in a less digestible format.

For now, let's just create some dummy text that we can use to test our scrollytelling so that writing the narrative is not a blocker:

```
// This is one example narrative you can create with each commit
<p>
  On {commit.datetime.toLocaleString("en", {dateStyle: "full", timeStyle:
  "short"})}, I made
  <a href="{commit.url}" target="_blank">
    { index > 0 ? 'another glorious commit' : 'my first commit, and it was glorious' }
  </a>. I edited {commit.totalLines} lines across { d3.rollups(commit.lines, D =>
  D.length, d => d.file).length } files. Then I looked over all I had made, and
  I saw that it was very good.
</p>
```

Now given this structure, hink about how you can set it as an attribute or further append as a child to the elements in `itemsContainer` using D3.

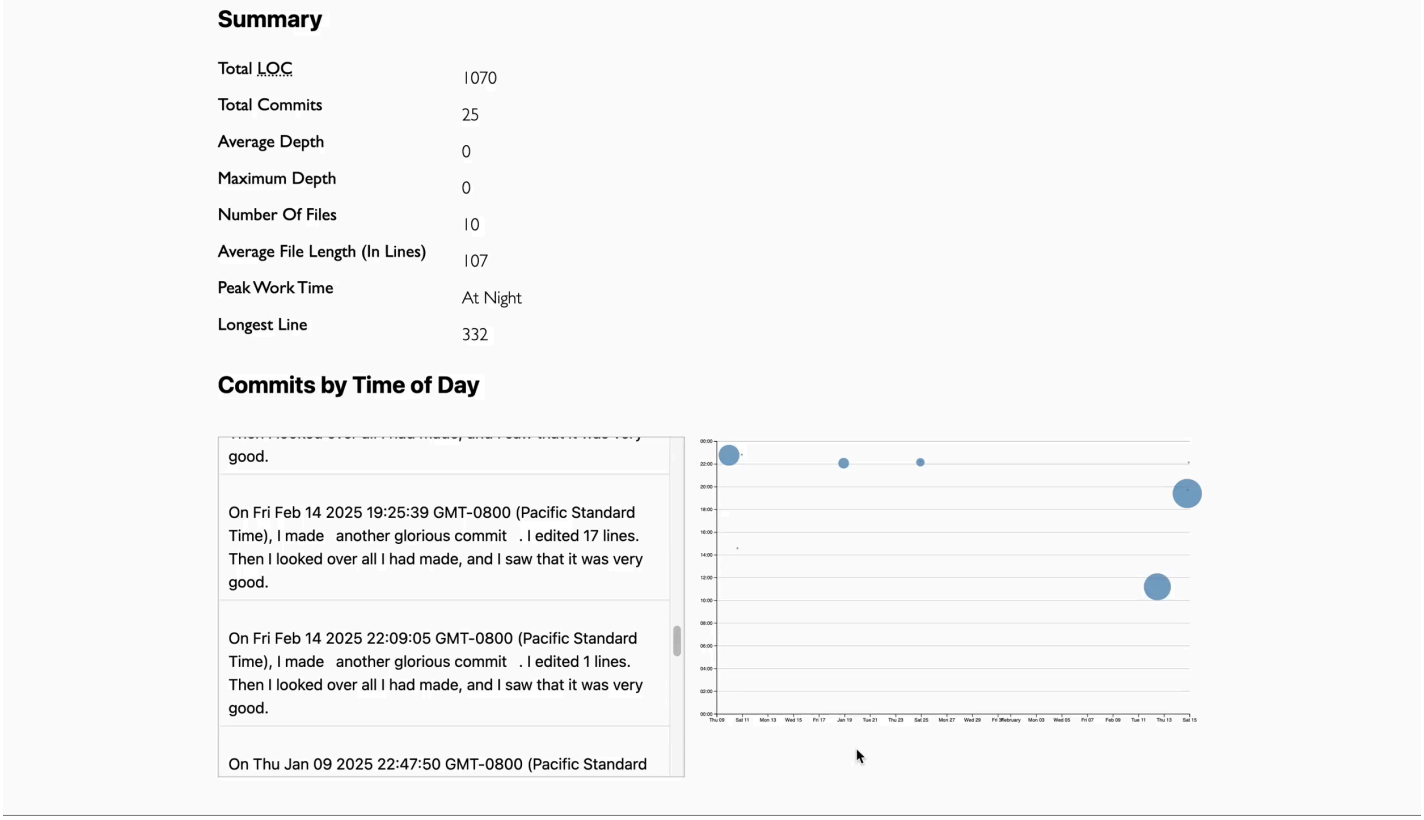
> **CAVEAT**
> If your narrative overflows, it's because we pre-set a really small item height `height: 30px;`. Play around with bigger values until it gives enough space for each of your narratives. And make sure to update the height in both the CSS rule and the `ITEM_HEIGHT` variable.

Step 4.3: Creating a scroller for our commits over time

Integrate the story you just generated into commit items slicing and rendering.

Also, notice how we were able to create a variable `newCommitSlice` that stores the commits rendered visible from the current scrolling event. We can convenient have our scatter plot to also update based on the same set of commits so they can be consistent!

Once you do that, you should see the following:



> **NOTE**
>
> Note that the summary stats stay unchanged in this case since they still capture the overall information about all your commits. You can make them also update with your scrollytelling. Just re-calculate the stats using `newCommitSlice`. But for simplicity just tie summary stats update to one of the scrollytellings (yes, we will make another one, keep reading Step 4).

Now that everything works, you should remove the slider as it is irrelevant/redundant with the scrolly, and it's largely repeating information that the scrollbar already provides. However, you do want to save some useful functionalities we implemented before about file sizes. Let's do the following:

```
function displayCommitFiles() {
  const lines = filteredCommits.flatMap((d) => d.lines);
  let fileTypeColors = d3.scaleOrdinal(d3.schemeTableau10);
  let files = d3.groups(lines, (d) => d.file).map(([name, lines]) => {
    return { name, lines };
  });
  files = d3.sort(files, (d) => -d.lines.length);
  d3.select('.files').selectAll('div').remove();
  let filesContainer = d3.select('.files').selectAll('div').data(files).enter().append('div');
```

```
    filesContainer.append('dt').html(d => `<code>${d.name}</code><small>${d.lines.length} lines</small>`);
    filesContainer.append('dd')
                  .selectAll('div')
                  .data(d => d.lines)
                  .enter()
                  .append('div')
                  .attr('class', 'line')
                  .style('background', d => fileTypeColors(d.type));
}
```

Now you may still be able to update the commit files in display by calling `displayCommitFiles()`. And you now safely remove code that handles event handling on the slider, as well as the HTML code themselves.

Also notice that the demo above has the scrolly and the scatter plot displayed side by side, think about how you can achieve the same using `grid` and `subgrid` displays.

> **GOING FURTHER**
> One thing you could do is show a date next to the actual browser scrollbar thumb, so that users have a sense of where they are in the timeline.

Step 4: Scrollytelling Part 2 (file sizes)

Step 4.1: Adding another scrolly

Create another scrolly for the file sizes visualization (e.g. how many lines you edited, think back to Step 2. You may directly edit your `src/meta/index.html` for set-up), after the commit visualization. You can copy and paste the same narrative as a temporary placeholder, but as with the one about commits, you should replace it with something meaningful before you finish the lab.

You will want to use a different variable for it since the filtering condition is likely different. Aside from that, the rest is very similar to Step 4. `displayCommitFiles()` should help make things easy for you.

To make it more visually interesting, you may try to place the scrolly on the right while the unit visualization on th left.

Resources

Transitions & Animations

Tech:

- Cheatsheet on animation-related technologies
- An interactive guide to CSS transitions

Scrollytelling

Cool examples:

- This is a teenager
- A visual introduction to Machine Learning Part 1
- A visual introduction to Machine Learning Part 2
- Ben & Jerry's