Front Ends

Single-Page Apps

- Single-page apps take content that would ordinarily be on multiple different pages (or routes) and combine them into a single page that pulls new information from the server whenever itâ€™s needed (through methods such as AJAX).
- For a starting point, this application uses multiple pages.

```
@app.route("/")
def first():
    return render_template("first.html")

@app.route("/second")
def second():
    return render_template("second.html")

@app.route("/third")
def third():
    return render_template("third.html")
```

- Hereâ€™s the layout template for these pages.

```
<html>
    <head>
        <title>My Webpage</title>
    </head>
    <body>
        <ul id="nav">
            <li><a href="">First Page</a></li>
            <li><a href="">Second Page</a></li>
            <li><a href="">Third Page</a></li>
        </ul>
        <hr>

        {% block body %}
        {% endblock %}

    </body>
</html>
```

  - The navigation bar is simply an unordered list of links.
- Given that these pages all have the simple function of displaying text, `application.py` can be reworked to run on a single route.

```
@app.route("/")
def index():
    return render_template("index.html")

texts = ["text 1", "text 2", "text 3"]

@app.route("/first")
def first():
    return texts[0]

@app.route("/second")
def second():
    return texts[1]

@app.route("/third")
def third():
    return texts[2]
```

  - Note that the other â€˜routesâ€™ donâ€™t return a new webpage, but rather just the text that should be displayed.
- In order to process this structure, JavaScript must be added to `index.html`.

```
<html>
    <head>
        <script>
            document.addEventListener('DOMContentLoaded', () => {

                // Start by loading first page.
                load_page('first');

                // Set links up to load new pages.
                document.querySelectorAll('.nav-link').forEach(link => {
                    link.onclick = () => {
                        load_page(link.dataset.page);
```

```
                        return false;
                    };
                });
            });

            // Renders contents of new page in main view.
            function load_page(name) {
                const request = new XMLHttpRequest();
                request.open('GET', `/${name}`);
                request.onload = () => {
                    const response = request.responseText;
                    document.querySelector('#body').innerHTML = response;
                };
                request.send();
            }
        </script>
    </head>
    <body>
        <ul id="nav">
            <li><a href="" class="nav-link" data-page="first">First Page</a></li>
            <li><a href="" class="nav-link" data-page="second">Second Page</a></li>
            <li><a href="" class="nav-link" data-page="third">Third Page</a></li>
        </ul>
        <hr>
        <div id="body">
        </div>
    </body>
</html>
```

  - `load_page` makes an AJAX request to the server to get the text that should be displayed and puts in the `body div`.
- This new single-page implementation avoids reloading the page repeatedly just to display very similar content (e.g. the same navigation bar). However, this eliminates the URLâ€™s functionality as a locator, because all the content is on the same route.

HTML5 History API

- The HTMuL5 History API allows for the manipulation of a browserâ€™s history and URL even if the page is still being implemented with a single-page design. Whenever an new â€˜pageâ€™ is accessed, the client can â€˜pushâ€™ a new URL state.
- The changes to the JavaScript code are inside the `load_page` function.

```
function load_page(name) {
    const request = new XMLHttpRequest();
    request.open('GET', `/${name}`);
    request.onload = () => {
        const response = request.responseText;
        document.querySelector('#body').innerHTML = response;

        // Push state to URL.
        document.title = name;
        history.pushState(null, name, name);
    };
    request.send();
}
```

  - `document.title` is just an aesthetic property that is set to reflect the current page.
  - In the `history.pushState()` function, which is used to change the browserâ€™s history, the first argument is any data that should be associated with the push, the second argument is the title of the page being pushed, and the third argument is the URL being pushed.
- One flaw with this, though, is that the full multi-page behavior is not truly being emulated. If a user tries to use the back button in their browser, the URL will change, but not the content. To remedy this, the full, stack-like behavior of the HTML5 History API can be used. Going back in history should just â€˜popâ€™ whatever the URL is on top off of the stack.

```
// Renders contents of new page in main view.
function load_page(name) {
    const request = new XMLHttpRequest();
    request.open('GET', `/${name}`);
    request.onload = () => {
        const response = request.responseText;
        document.querySelector('#body').innerHTML = response;

        // Push state to URL.
        document.title = name;
        history.pushState({'title': name, 'text': response}, name, name);
    };
    request.send();
}

// Update text on popping state.
```

```
window.onpopstate = e => {
    const data = e.state;
    document.title = data.title;
    document.querySelector('#body').innerHTML = data.text;
};
```

- Now, when pushing a new state, title and text data is being pushed with it.
- When the state is popped, `e`, the event that just took place, has a `state` property that contains all the data that was pushed with that state. Then, that data is just used to update the contents of the page as expected.

Window and Document

- The `window` and `document` variables, which have been seen in past examples, are just examples of JavaScripts objects on which operations can be performed and that have properties that can be accessed. In particular, they contain information about their size and position.
  - `window.innerWidth` : window width
  - `window.innerHeight` : window height
  - `document.body.offsetHeight` : the entire height of the HTML bodyâ€™s document, of which the window height is likely just a small porition
  - `window.scrollY` : how far down the page has been scrolled (in pixels)
- One potential use of these properties is to be able to detect if the user has scrolled to the bottom of the page.

```
window.onscroll = () => {
    console.log('----');
    console.log(window.innerHeight);
    console.log(window.scrollY);
    console.log(document.body.offsetHeight);
    if (window.innerHeight + window.scrollY >= document.body.offsetHeight) {
        document.querySelector('body').style.background = 'green';
    } else {
        document.querySelector('body').style.background = 'white';
    }
};
```

  - `console.log` is essentially a print statement that prints to the web browsersâ€™s console.
  - All this does is change the background color of the web page to green when the bottom of the document has been reached, which is detected using the mathematical relationship between `window` and `document` properties.
- A more useful application of this bottom-detection would be the dynamic loading of more content when the bottom of a webpage has been reached. `application.py` for such a webpage could look like this.

```
import time

from flask import Flask, jsonify, render_template, request

app = Flask(__name__)

@app.route("/")
def index():
    return render_template("index.html")

@app.route("/posts", methods=["POST"])
def posts():

    # Get start and end point for posts to generate.
    start = int(request.form.get("start") or 0)
    end = int(request.form.get("end") or (start + 9))

    # Generate list of posts.
    data = []
    for i in range(start, end + 1):
        data.append(f"Post #{i}")

    # Artificially delay speed of response.
    time.sleep(1)

    # Return list of posts.
    return jsonify(data)
```

- `index.html` (a little more complex now)

```
<html>
    <head>
        <script>
            // Start with first post.
            let counter = 1;
```

```
                // Load posts 20 at a time.
                const quantity = 20;

                // When DOM loads, render the first 20 posts.
                document.addEventListener('DOMContentLoaded', load);

                // If scrolled to bottom, load the next 20 posts.
                window.onscroll = () => {
                    if (window.innerHeight + window.scrollY >= document.body.offsetHeight) {
                        load();
                    }
                };

                // Load next set of posts.
                function load() {

                    // Set start and end post numbers, and update counter.
                    const start = counter;
                    const end = start + quantity - 1;
                    counter = end + 1;

                    // Open new request to get new posts.
                    const request = new XMLHttpRequest();
                    request.open('POST', '/posts');
                    request.onload = () => {
                        const data = JSON.parse(request.responseText);
                        data.forEach(add_post);
                    };

                    // Add start and end points to request data.
                    const data = new FormData();
                    data.append('start', start);
                    data.append('end', end);

                    // Send request.
                    request.send(data);
                };

                // Add a new post with given contents to DOM.
                function add_post(contents) {

                    // Create new post.
                    const post = document.createElement('div');
                    post.className = 'post';
                    post.innerHTML = contents;

                    // Add post to DOM.
                    document.querySelector('#posts').append(post);
                };
            </script>
        </head>
        <body>
            <div id="posts">
            </div>
        </body>
    </html>
```

- For a little more functionality, the `add_post` function could be modified to add a button to hide uninteresting posts.

```
function add_post(contents) {

    // Create new post.
    const post = document.createElement('div');
    post.className = 'post';
    post.innerHTML = contents;

    // Add button to hide post.
    const hide = document.createElement('button');
    hide.className = 'hide';
    hide.innerHTML = 'Hide';
    post.append(hide);

    // When hide button is clicked, remove post.
    hide.onclick = function() {
        this.parentElement.remove();
    };

    // Add post to DOM.
    document.querySelector('#posts').append(post);
};
```

- Calling `post.append(hide)` adds the hide button inside the post `div`.
- `parentElement` is the element containing the element in question. In this case, `this.parentElement` is used to refer to the `post` containing the `hide` button.
- `remove` is a built-in function to delete an element all together.

JavaScript Templating

- One issue with using JavaScript to build more complicated user interfaces and adding items to the DOM the code is starting to get a little bit messy. Every element needs to be created, class names need to be assigned, inner HTML needs to be set, etc. Ideally, all the HTML would be written somewhere else, but the exact content thatâ€™s going inside is still currently unknown.
- The solution to this problem is JavaScript templating, which allows for the creation of templates in JavaScript that define the HTML, while also allowing for substitution inside that template for adding different content. A very simple version of this is JavaScriptâ€™s template literals. There many different JavaScript libraries that take that idea one step further. In this class, the Handlebars library will be used.
- The next series of examples will be a dice-throwing application. Hereâ€™s the starting point.

```
<html>
    <head>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/4.0.11/handlebars.min.js"></script>
        <script>
            // Template for roll results
            const template = Handlebars.compile("<li>You rolled a </li>");

            document.addEventListener('DOMContentLoaded', () => {
                document.querySelector('#roll').onclick = ()  => {

                    // Generate a random roll.
                    const roll = Math.floor((Math.random() * 6) + 1);

                    // Add roll result to DOM.
                    const content = template({'value': roll});
                    document.querySelector('#rolls').innerHTML += content;
                };
            });
        </script>
    </head>
    <body>
        <button id="roll">Roll</button>
        <ul id="rolls">
        </ul>
    </body>
</html>
```

  - `template` is being used repeatedly for every roll. It is like a client-side analog to the Flask/Jinja2 templates.
  - `Math.random()` returns a random number between 0 and 1. Multiplying it by 6 returns a number in the range of 0 up to, but not including, 6. Adding 1 gives a range from 1 up to 7, and using `Math.floor()` will return either 1, 2, 3, 4, 5, or 6.
  - `template` is used like function: it is passed value(s) and returns HTML content.
- It would be nicer to have images of the dice roll rather than just printing out the number. To do so, all that needs to change is the template, which now includes an `img` element.

```
const template = Handlebars.compile("<li>You rolled: <img src=\"img/.png\"></li>");
```

  - Note how the `"` characters are escaped, since they are inside a string.
- Still, including all of the JavaScript template inside a string starts to get messy when including images, etc. Ideally, there would be pure HTML that is then compiled by Handlebars.

```
<script id="result" type="text/x-handlebars-template">
    <li>
        You rolled:

        <img alt="{{ value }}" title="{{ value }}" src="img/{{ value }}.png"></img>

    </li>
</script>
<script>
    // Template for roll results
    const template = Handlebars.compile(document.querySelector('#result').innerHTML);

    document.addEventListener('DOMContentLoaded', () => {
        document.querySelector('#roll').onclick = ()  => {

            // Generate a random roll.
            const roll = Math.floor((Math.random() * 6) + 1);

            // Add roll result to DOM.
```

```
            const content = template({'value': roll});
            document.querySelector('#rolls').innerHTML += content;
        };
    });
</script>
```

- Note that there are two `script` elements. The one with the id `result` with represent the result of a roll. It has a special `type` attribute, defined by Handlebars. Inside of this `script` element will be HTML code that represents the Handlebars template.
- The `alt` and `title` attributes of the image simply provide the same information in text when the image is hovered over and for browsers that donâ€™t support images.
- Now, instead of compiling a string, the template is simply selected using `document.querySelector`.

- Handlebars, like Jinja, supports loops. In this example, loops could be used to roll multiple dice at once.

```
<html>
    <head>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/4.0.11/handlebars.min.js"></script>
        <script id="result" type="text/template">
            <li>
                You rolled:

                {{#each values}}
                    <img alt="{{ this }}" title="{{ this }}" src="img/{{ this }}.png">
                {{/each}}
                (Total: {{ total }})

            </li>
        </script>
        <script>

            // Template for roll results
            const template = Handlebars.compile(document.querySelector('#result').innerHTML);

            document.addEventListener('DOMContentLoaded', () => {
                document.querySelector('#roll').onclick = ()  => {

                    // Generate random rolls.
                    const counter = parseInt(document.querySelector('#counter').value);
                    const rolls = [];
                    let total = 0;
                    for (let i = 0; i < counter; i++) {
                        const value = Math.floor(Math.random() * 6) +  1;
                        rolls.push(value);
                        total += value;
                    };

                    // Add roll results to DOM.
                    const content = template({'values': rolls, 'total': total});
                    document.querySelector('#rolls').innerHTML += content;
                };
            });
        </script>
    </head>
    <body>
        <input id="counter" type="number" placeholder="Number of Rolls" min="1" value="1">
        <button id="roll">Roll</button>
        <ul id="rolls">
        </ul>
    </body>
</html>
```

- `#each` is a Handlebars â€˜block helperâ€™. There many of these helpers with different functions, be it loops, in this examples, conditionals (`#if`), etc. If the built-in helpers arenâ€™t enough, Handlebars also allows for the creation of custom helpers.
- Inside the loop, Handlebars calls every item in the set of items (in this case, the set is called `values`), `this`.

- One thing to keep in mind when adding Handlebars templates to Flask apps is that Jinja will scan the HTML file first, and will see the double curly brace syntax as a place where it should plug in a value. Since this is not desired, Jinja needs to be told to ignore the blocks of code with Handlebars templates with Jinjaâ€™s `raw` block.

```
{% raw -%}
    {{ contents }}
{%- endraw %}
```

CSS Animation

- CSS animation allows for changes from one CSS property to another over some duration of time while the page is running.

```
@keyframes grow {
    from {
        font-size: 20px;
    }
    to  {
        font-size: 100px;
    }
}


h1 {
    animation-name: grow;
    animation-duration: 2s;
    animation-fill-mode: forwards;
}
```

  - `@keyframes grow` defines a CSS animation called `grow`, which goes `from` one style `to` another style.
  - The `animation-name` property is used to link the `grow` animation to `h1` elements.
  - `animation-duration` sets the time over which the animation occurs.
  - `animation-fill-mode` sets the direction the animation should go. The value `forwards` means that once the end of the animation is reached, that final styling should be preserved.
- Another simple example:

```
@keyframes move {
    from {
        left: 0%;
    }
    to  {
        left: 50%;
    }
}

h1 {
    position: relative;
    animation-name: move;
    animation-duration: 3s;
    animation-fill-mode: forwards;
}
```

  - `left` indicates the relative position of an HTML element. `h1` is given the `position: relative` property, which means it position is defined in relationship to other parts of the window.
  - The `move` animation shifts an element from being 0% away from the left edge of the screen to being 50% away from that edge (aligned with the middle of the window).
- Along with a start and end point, midway points can be specified as well.

```
@keyframes move {
    0% {
        left: 0%;
    }
    50% {
        left: 50%;
    }
    100% {
        left: 0%;
    }
}
```

Adding JavaScript

- With CSS alone, animations always run as soon as a webpage is loaded. To control animation, JavaScript can be used to modify the CSS properties `animationPlayState`, which is `paused` or `running`.

```
<style>
    @keyframes move {
        0% {
            left: 0%;
        }
        50% {
            left: 50%;
        }
```

```
            100% {
                left: 0%;
            }
        }

        h1 {
            position: relative;
            animation-name: move;
            animation-duration: 3s;
            animation-fill-mode: forwards;
            animation-iteration-count: infinite;
        }

    </style>
    <script>
        document.addEventListener('DOMContentLoaded', () => {
            const h1 = document.querySelector('h1');
            h1.style.animationPlayState  = 'paused';
            document.querySelector('button').onclick = () => {
                if (h1.style.animationPlayState  === 'paused')
                    h1.style.animationPlayState = 'running';
                else
                    h1.style.animationPlayState  = 'paused';
            };
        });
    </script>
```

  - `animation-iteration-count` specifies how many times the animation should be run.
  - When the page is first loaded, the animation is paused. Then, everytime some button is clicked, the `animationPlayState` is changed.
- So far, animation has been purely aesthetic, but it can be a large part of a good user interface. One such situation might be the previous example with a list of posts. When hiding a post, it would helpful to have the post fade away.

```
    <style>

        @keyframes hide {
            from {
                opacity: 1;
            }
            to {
                opacity: 0;
            }
        }

        .post {
            background-color: #77dd11;
            padding: 20px;
            margin-bottom: 10px;
            animation-name: hide;
            animation-duration: 2s;
            animation-fill-mode: forwards;
            animation-play-state: paused;
        }
    </style>
    <script>
        // ...rest of JavaScript code...

        // If hide button is clicked, delete the post.
        document.addEventListener('click', event => {
            const element = event.target;
            if (element.className === 'hide') {
                element.parentElement.style.animationPlayState = 'running';
                element.parentElement.addEventListener('animationend', () =>  {
                    element.parentElement.remove();
                });
            }
        });
    </script>
```

  - There is slightly different logic here to figure out when the button is clicked. Now, anytime a mouse click occurs, the `event.target`, which is the element being clicked, is assigned to the variable `element`.
  - If the hide button was clicked, the animation is run on the post, and the end of the animation is listened for with a callback to actually delete the post.
- A slight refinement would be to have the rest of the posts slide up to fill the gap left by the deleted post. To give this illusion, only the actual post being deleted needs to have its animation modified, not all the posts remaining.

```
@keyframes hide {
    0% {
        opacity: 1;
        height: 100%;
        line-height: 100%;
        padding: 20px;
        margin-bottom: 10px;
    }
    75% {
        opacity: 0;
        height: 100%;
        line-height: 100%;
        padding: 20px;
        margin-bottom: 10px;
    }
    100% {
        opacity: 0;
        height: 0px;
        line-height: 0px;
        padding: 0px;
        margin-bottom: 0px;
    }
}
```

- For the first 75% of the animation, the post disappears.
- For the final 25% of the animation, the post shrinks in size until it has no height, causing all the other posts below it to fill in that space.

SVG Animation

- A Scalable Vector Graphic (SVG) is graphical element determined by lines, angles, and shapes. SVGs can be used to draw things that simple HTML elements, like `divs`, donâ€™t allow for.

```
<body>
    <svg style="width:100%; height:800px">
        <circle cx="200" cy="200" r="50" style="fill:blue"/>
    </svg>
</body>
```

  - The SVG element is given a fixed `height` and a `width` that automatically adjusts based on the content to maintain that fixed height.
  - The `circle` element is one of the SVG elements supported by SVG. It is given x- and y-coordinates for its center with `cx` and `cy`, a radius with `r`, and finally some CSS styling.
- As before, it is preferable to be able to create such elements programatically using JavaScript. To do so, a JavaScript data visualization library, D3, will be used.

```
<html>
    <head>
        <script src="https://d3js.org/d3.v4.min.js"></script>
    </head>
    <body>
        <svg id="svg" style="width:100%; height:800px"/>
    </body>
    <script>

        const svg = d3.select('#svg');

        svg.append('circle')
           .attr('cx', 200)
           .attr('cy', 200)
           .attr('r', 90)
           .style('fill', 'green');

    </script>
</html>
```

  - `d3.select` gets access to an HTML element.
  - Then, D3 functions are used to add a circle to that selected SVG element with all the same attributes and styling as before.
- As with CSS, animations can be added to SVGs.

```
const svg = d3.select('#svg');

const c = svg.append('circle')
             .attr('cx', 200)
             .attr('cy', 200)
```

```
                .attr('r', 50)
                .style('fill', 'blue');

c.transition()
 .duration(1000)
 .attr('cx', 500)
 .attr('cy', 500)
 .style('fill', 'red');
```

  - The duration (in milliseconds) for the `transition` (animation) is given, along with the final values for each attribute that should be animated.
- Animations can also delayed or triggered on certain events.

```
c.transition()
 .duration(1000)
 .delay(1000)
 .attr('cx', 500);

c.on('click', function() {
    d3.select(this).transition()
                   .duration(3000)
                   .style('fill', 'red');
});
```

  - `delay` specifies the length of time before the animation is run.
  - `on` takes an event and callback to apply to an SVG. In this case, when the circle is clicked, `this`, whatever was clicked on, undergoes another transition.

A Drawing Application

- The final example of a user interface, demonstrating the potential of SVGs, will be a simple sketchpad-like application.

```
<body>
    <svg id="svg" style="width:100%; height:800px"/>
</body>
<script>

    const svg = d3.select('#svg');

    function draw_point() {
        const coords = d3.mouse(this);

        svg.append('circle')
           .attr('cx', coords[0])
           .attr('cy', coords[1])
           .attr('r', 5)
           .style('fill', 'black');
    };

    svg.on('mousemove', draw_point);

</script>
```

  - Whenever the mouse moves on the canvas, `draw_point` will be called.
  - `draw_point` simply draws a small circle where the mouse is, grabbing its coordinates with `d3.mouse(this)`.
- An obvious improvement would be to only draw when the mouse is clicked.

```
const svg = d3.select('#svg');
let drawing = false;

function draw_point() {
    if (!drawing)
        return;

    const coords = d3.mouse(this);

    svg.append('circle')
       .attr('cx', coords[0])
       .attr('cy', coords[1])
       .attr('r', 5)
       .style('fill', 'black');
};

svg.on('mousedown', () => {
    drawing = true;
});

svg.on('mouseup', () => {
```

```
    drawing = false;
});

svg.on('mousemove', draw_point);
```

  - Now, a boolean variable `drawing` controls whether or not a point should be drawn.
  - Clicking (`mousedown`) turns on drawing by setting `drawing` to `true`, and releasing `mouseup` turns it off.
- The remaining problem is that if the mouse moves too fast, a bunch of unconnected dots will be drawn because the `mousemove` event isnâ€™t fired quickly enough. This frequency cannot be changed, but one workaround would be to draw a line between all points.
- First off, a nicer UI would include a list of options to let the user choose pen color, thickness, and also to erase the canvas.

```
<body>
    <div class="container">
        <div id="options" class="row">
            <select id="color-picker">
                <option value="black">Black</option>
                <option value="red">Red</option>
                <option value="blue">Blue</option>
                <option value="green">Green</option>
            </select>
            <select id="thickness-picker">
                <option value=1>1</option>
                <option value=2>2</option>
                <option value=3 selected>3</option>
                <option value=4>4</option>
                <option value=5>5</option>
                <option value=6>6</option>
                <option value=7>7</option>
                <option value=8>8</option>
                <option value=9>9</option>
                <option value=10>10</option>
            </select>
            <button id="erase">Erase</button>
        </div>
    </div>
    <svg id="draw">
    </svg>
</body>
```

- The more complex JavaScript now takes into account these features.

```
document.addEventListener('DOMContentLoaded', () => {

    // state
    let draw = false;

    // elements
    let points = [];
    let lines = [];
    let svg = null;

    function render() {

        // create the selection area
        svg = d3.select('#draw')
                .attr('height', window.innerHeight)
                .attr('width', window.innerWidth);

        svg.on('mousedown', function() {
            draw = true;
            const coords = d3.mouse(this);
            draw_point(coords[0], coords[1], false);
        });

        svg.on('mouseup', () =>{
            draw = false;
        });

        svg.on('mousemove', function() {
            if (!draw)
                return;
            const coords = d3.mouse(this);
            draw_point(coords[0], coords[1], true);
        });

        document.querySelector('#erase').onclick = () => {
            for (let i = 0; i < points.length; i++)
                points[i].remove();
            for (let i = 0; i < lines.length; i++)
```

```
                lines[i].remove();
            points = [];
            lines = [];
        }

    }

    function draw_point(x, y, connect) {

        const color = document.querySelector('#color-picker').value;
        const thickness = document.querySelector('#thickness-picker').value;

        if (connect) {
            const last_point = points[points.length - 1];
            const line = svg.append('line')
                            .attr('x1', last_point.attr('cx'))
                            .attr('y1', last_point.attr('cy'))
                            .attr('x2', x)
                            .attr('y2', y)
                            .attr('stroke-width', thickness * 2)
                            .style('stroke', color);
            lines.push(line);
        }

        const point = svg.append('circle')
                         .attr('cx', x)
                         .attr('cy', y)
                         .attr('r', thickness)
                         .style('fill', color);
        points.push(point);
    }

    render();
});
```

- All points and lines are saved in arrays to allow them to be cleared when the user erases the canvas.
- Now, draw_point takes three arguments: the coordinates of the point and whether it should be connected to the previous point. It should not be connected when the mouse is clicked for the first time, but it should be connected whenever the mouse is moved.
- The `draw_point` function grabs the selected color and thickness, and, if the last point should be connected, it also grabs that point. A line with endpoints at the old point and the mouse location is then drawn and added to the array `lines`.
- The point is drawn as before, but also added to the array `points`.