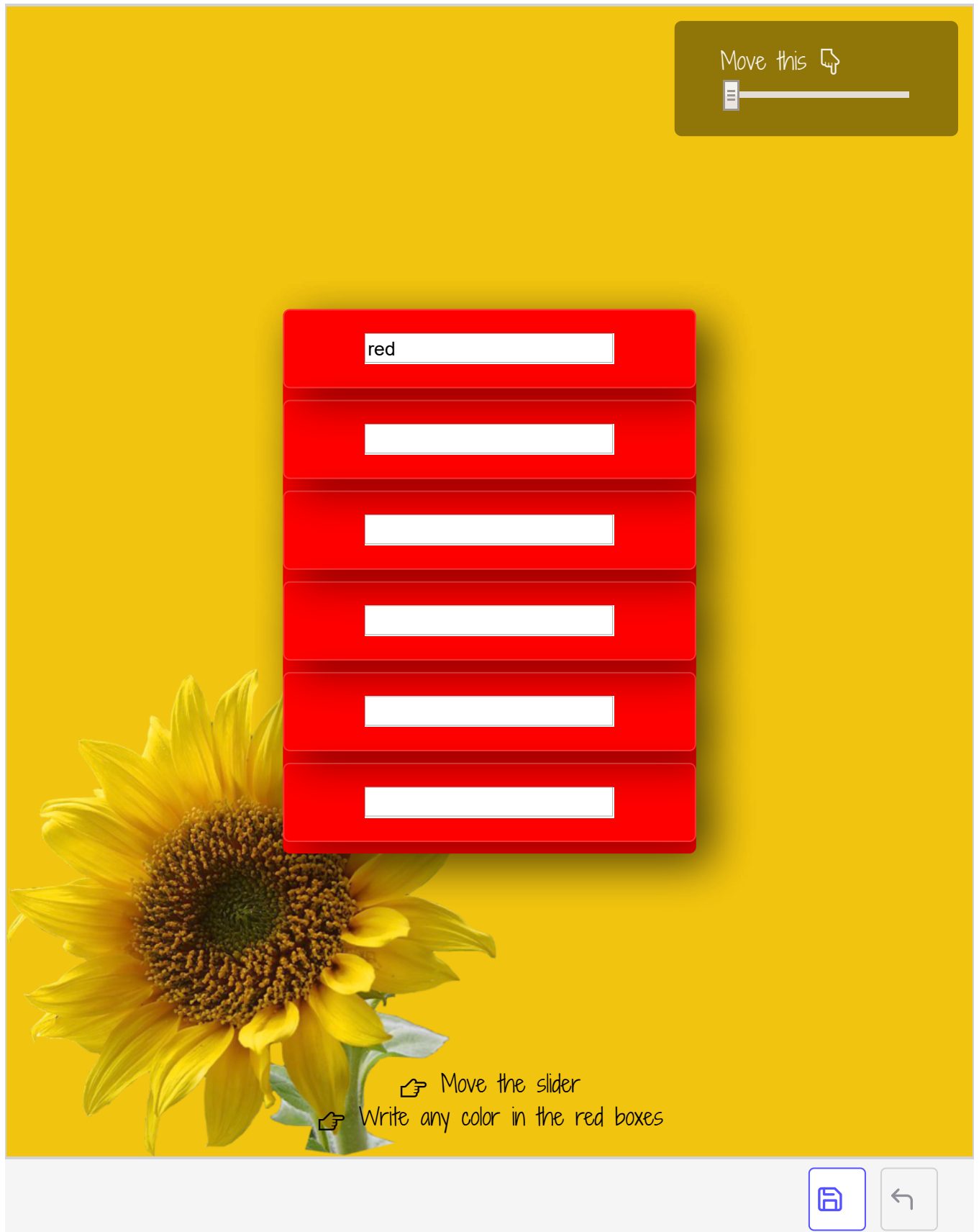


Building the CSS Variable Booth 🎨

In case you missed it, below is what we'll build.

Type any color values within the red boxes, and move the range slider on the top right too!

Output
JavaScript
HTML
CSS (SCSS)



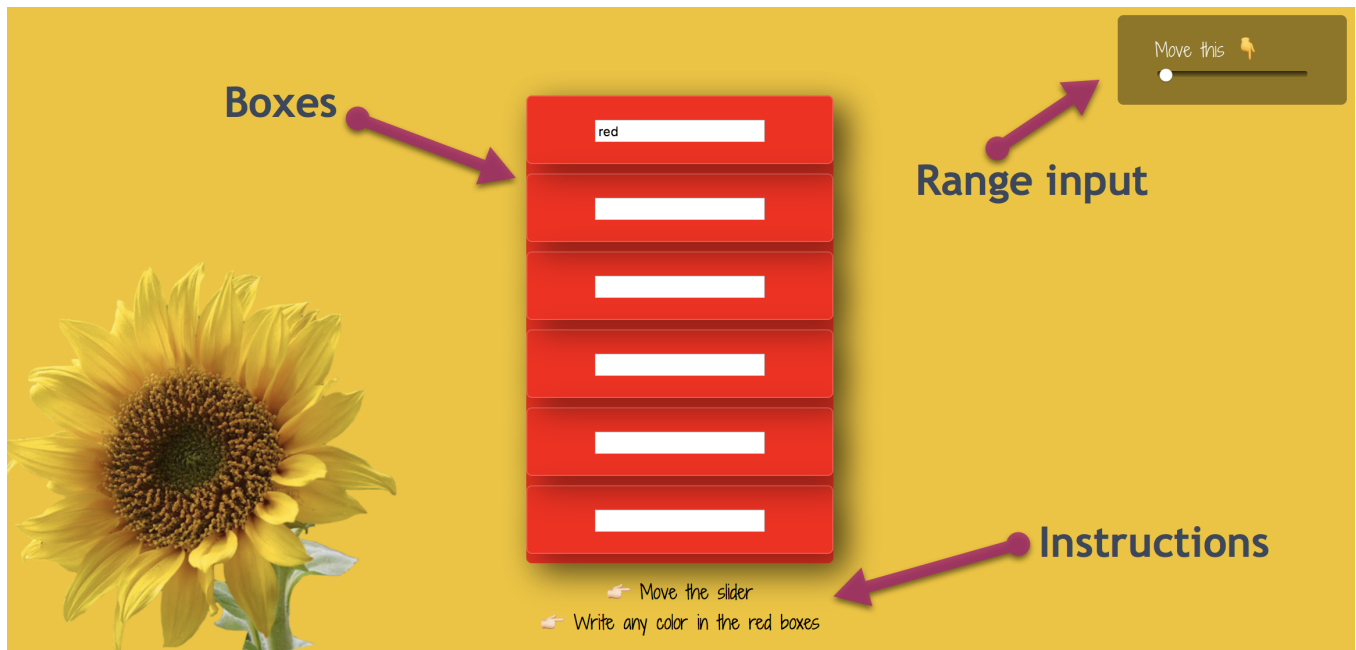
This is a superb example of updating CSS variables with Javascript and the reactivity that comes with it.

Let's see how to build this.

The Markup


Here are the necessary components.

1. A range input
2. A container to hold the instructions
3. A section to hold a list of other boxes each containing input fields





The markup turns out to be simple.

Here It is:

```
<main class="booth">
  <aside class="slider">
    <label>Move this  </label>
    <input class="booth-slider" type="range" min="-50" max="50" value="-50" step="5"/>
  </aside>

  <section class="color-boxes">
    <div class="color-box" id="1"><input value="red"/></div>
    <div class="color-box" id="2"><input/></div>
    <div class="color-box" id="3"><input/></div>
    <div class="color-box" id="4"><input/></div>
    <div class="color-box" id="5"><input/></div>
    <div class="color-box" id="6"><input/></div>
  </section>

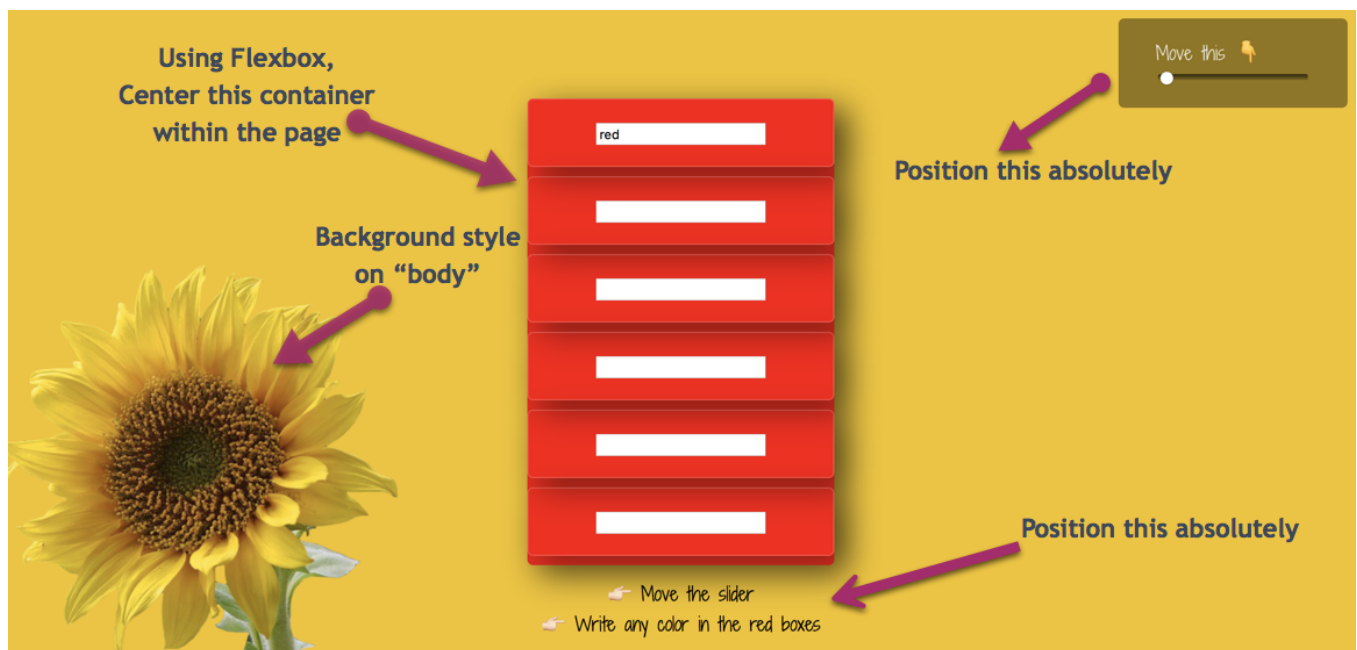
  <footer class="instructions">
     Move the slider<br/>
     Write any color in the red boxes
```

```
    Write any color in the red boxes  
</footer>  
</main>
```

Here a few things to point your attention to.

1. The range input represents values from `-50` to `50` with a step value of `5`. Also, the value of the range input is the minimum value, `-50`. If you aren't sure how the range input works, check it out [on w3schools](#)
2. Note how the section with class `.color-boxes` contains other `.color-box` containers. Within these containers are input fields. It is perhaps worth mentioning that the first input has a default value of red.

Having understood the structure of the document, go ahead and style it like so:



1. Take the `.slider` and `.Instructions` containers out of the document flow. Position them absolutely.
2. Give the `body` element a sunrise background color and garnish the background with a flower in the bottom left corner
3. Position the `color-boxes` container in the center
4. Style the `color-boxes` container

Let's knock these off.

The following will fix the the first task.

```

/* Slider */
.slider,
.instructions {
  position: absolute;
  background: rgba(0,0,0,0.4);
  padding: 1rem 2rem;
  border-radius: 5px
}
.slider {
  right: 10px;
  top: 10px;
}
.slider > * {
  display: block;
}

/* Instructions */
.instructions {
  text-align: center;
  bottom: 0;
  background: initial;
  color: black;
}

```

The code snippet isn't as complex as you may think. I hope you can read through and understand it. If not, drop a comment or tweet.

Styling the `body` is a little more involved. Hopefully, you understand CSS well.

Since we aspire to style the element with a background color and a background image, it's perhaps the best bet to use the `background` shorthand property to set multiple backgrounds.

Here it is:

```

body {
  margin: 0;
  color: rgba(255,255,255,0.9);
}

```

```
background: url('http://bit.ly/2FiPrRA') 0 100%/340px no-repeat, var(--primary-color);

font-family: 'Shadows Into Light Two', cursive;
}
```

The `url` bit is the link to the sunrise flower.

The next set of properties `0 100%` represent the background position of the image.

I hope you haven't forgotten the lessons on how the background shorthand works :)

The other bit after the forward slash represents the `background-size`. This has been set to `340px`. If you made this smaller, the image would be smaller too.

`no-repeat`, everyone knows what that does. It prevents the background from repeating itself.

Finally, anything that comes after the comma is a second background declaration. This time we've only set the `background-color` to `var(primary-color)`.

Oops, that's a variable.

The implication of this is that you have to define the variable. Here's how:

```
:root {
  --primary-color: rgba(241,196,15 ,1)
}
```

The primary color there is the sunrise yellow color. No big deal. We'll set some more variables in there soon.

Now, let's center the `color-boxes`.

```
main.booth {
  min-height: 100vh;
  display: flex;
```

```
display: flex;
justify-content: center;
align-items: center;
}
```

The main container acts as a flex container and correctly positions the direct child in the center of the page. This happens to be our beloved `color-box` container

Let's make the color-boxes container and its child elements pretty.

First, the child elements...

```
.color-box {
  padding: 1rem 3.5rem;
  margin-bottom: 0.5rem;
  border: 1px solid rgba(255,255,255,0.2);
  border-radius: 0.3rem;
  box-shadow: 10px 10px 30px rgba(0,0,0,0.4);
}
```

That will do it. There's a beautiful shadow added too. That'll give us a cool effect.

That is not all. Let's style the overall `container-boxes` container

```
/* Color Boxes */
.color-boxes {
  background: var(--secondary-color);
  box-shadow: 10px 10px 30px rgba(0,0,0,0.4);
  border-radius: 0.3rem;

  transform: perspective(500px) rotateY( calc(var(--slider) * 1deg));
  transition: transform 0.3s
}
```

Oh my!

There's a lot in there.

Let me break it down.

Here's the simple bit:

```
.color-boxes {  
  background: var(--secondary-color);  
  box-shadow: 10px 10px 30px rgba(0,0,0,0.4);  
  border-radius: 0.3rem;  
}
```

You know what that does, huh?

There's a new variable in there, whose syntax should be written as:

```
:root {  
  --primary-color: rgba(241,196,15 ,1);  
  --secondary-color: red;  
}
```


The secondary color is red. This will give the container a red background.

Now to the part that confused you.

```
/* Color Boxes */  
.color-boxes {  
  transform: perspective(500px) rotateY( calc(var(--slider) * 1deg));  
  transition: transform 0.3s  
}
```

For a moment, we could simplify the transform property value above.

```
.color-boxes {  
  transform: perspective(500px) rotateY(calc(var(--slider) * 1deg));  
}
```



perspective(500px) rotateY(60deg)


```
transform: perspective(500px) rotateY( 30deg);
```

The transform shorthand applies two different functions. One, the perspective, and the other, the rotation along the Y axis.

Hmmm, so what's the deal with the `perspective` and `rotateY` functions?

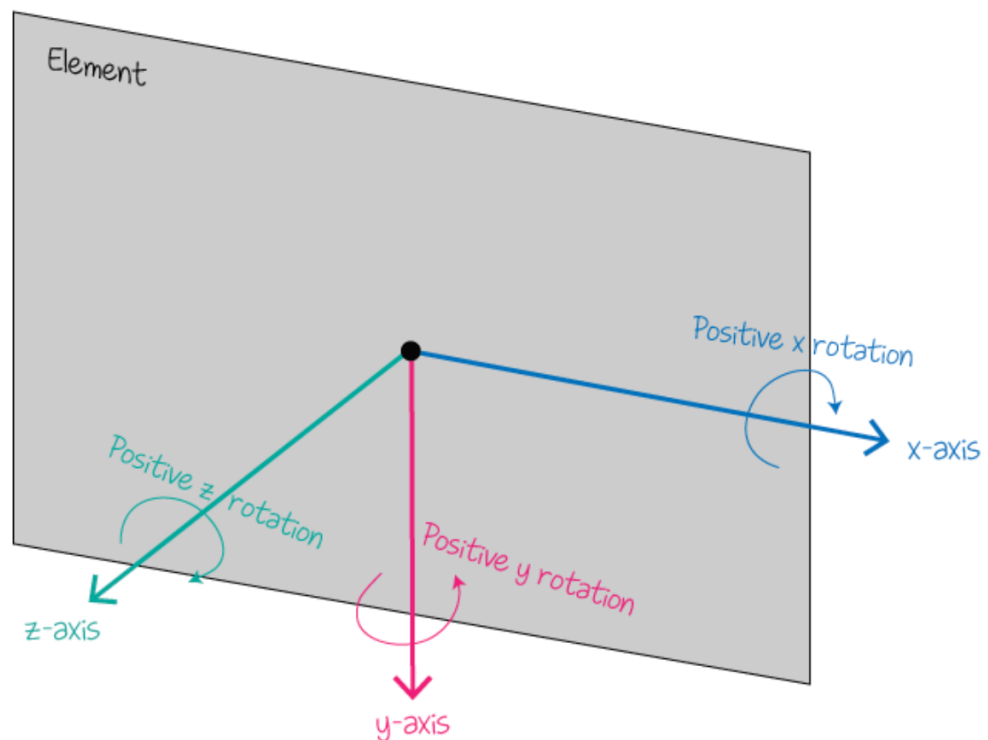
The `perspective()` function is applied to an element that is being transformed in 3D space. It activates the three dimensional space and gives the element depth along the z-axis.

You can read more about the perspective function on [codrops](#).

The `rotateY` function, what's the deal with that?

Upon activation the 3d space, the element has the planes x, y, z. The `rotateY` function rotates the element along the `Y` plane.

The following diagram from [codrops](#) is really helpful for visualizing this.



The positive direction of rotation along the three axes. Notice how if you stand at the end of each vector and look towards the origin, the clockwise rotation matches the one shown in the image.

Back to where we started:

```
transform: perspective(500px) rotateY( 30deg);
```

When you move the slider, do you know what function affects the rotation of the `.container-box` ?

Obviously, it's the `rotateY` function being invoked. The box is rotated along the Y axis.

Since the value passed into the `rotateY` function will be updated via Javascript, the value is represented with a variable

```
.color-boxes {  
  transform: perspective(500px) rotateY( calc(var(--slider) * 1deg));  
}
```

CSS variable

So, why multiply by the variable by `1deg`?

As a general rule of thumb, and for explicit freedom, it is advised that when building single tokens you store values in your variables without units.

You can convert them to any unit you want by doing a multiplication via the `calc` function

```
.color-boxes {  
  transform: perspective(500px) rotateY( calc(var(--slider) * 1deg));  
}
```

Plain number

This allows you to do ‘whatever’ you want with these values when you have them. Want to convert to `deg` or as a ratio of the user’s viewport `vw`, you can whatever you want.

In this case, we are converting the number to degrees by multiplying the “number” value by `1deg`

```
.color-boxes {  
  transform: perspective(500px) rotateY( calc(var(--slider) * 1deg));  
}
```

For example: `calc(30 * 1deg) = 30deg`

Since CSS doesn't understand math, you have to pass this arithmetic into the calc function to be properly evaluated by CSS.

Once that is done, we're good to go. The value of this variable can be updated in Javascript as often as we like.

Now, there's just one bit of CSS remaining.

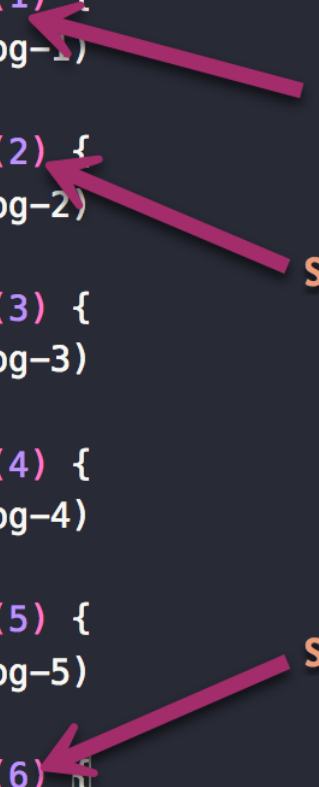
Here it is:

```
/* Handle colors for each color box */
.color-box:nth-child(1) {
  background: var(--bg-1)
}
.color-box:nth-child(2) {
  background: var(--bg-2)
}
.color-box:nth-child(3) {
  background: var(--bg-3)
}
.color-box:nth-child(4) {
  background: var(--bg-4)
}
.color-box:nth-child(5) {
  background: var(--bg-5)
}
.color-box:nth-child(6) {
  background: var(--bg-6)
}
```

So, what's this voodoo?

First off, the nth-child selector selects each of the child boxes.

```
.color-box:nth-child(1) {  
  background: var(--bg-1)  
}  
.color-box:nth-child(2) {  
  background: var(--bg-2)  
}  
.color-box:nth-child(3) {  
  background: var(--bg-3)  
}  
.color-box:nth-child(4) {  
  background: var(--bg-4)  
}  
.color-box:nth-child(5) {  
  background: var(--bg-5)  
}  
.color-box:nth-child(6) {  
  background: var(--bg-6)  
}
```



The diagram shows three pink arrows pointing from text labels to specific lines in the CSS code. The first arrow points from "Select box 1" to the selector `.color-box:nth-child(1)`. The second arrow points from "Select box 2" to the selector `.color-box:nth-child(2)`. The third arrow points from "Select box 6" to the selector `.color-box:nth-child(6)`.

There's a bit of foresight needed here. We know we will be updating the background color of each box. We also know that this background color has to be represented by a variable so it is accessible via javascript. Right?

We could go ahead and do this:

```
.color-box:nth-child(1) {  
  background: var(--bg-1)  
}
```

Easy.

There's one problem though. If this variable isn't present what happens? We need a fallback.

This works:

```
.color-box:nth-child(1) {  
  background: var(--bg-1, red)  
}
```

In this particular case, I have chosen to NOT provide any fallbacks.

If a variable used within a property value is invalid, the property will take on its initial value.

Consequently, when `--bg-1` is invalid or NOT available, the background will default to its initial value of transparent.

Initial values refer to the values of a property when they aren't explicitly set. If you don't set the `background-color` of an element, it will default to `transparent`. Initial values are kind of default property values.

Let's write some Javascript

There's very little we need to do on the Javascript side of things.

First let's handle the slider.

We just need 5 lines for that!

```
const root = document.documentElement  
const range = document.querySelector('.booth-slider')  
  
//as slider range's value changes, do something  
range.addEventListener('input', handleSlider)  
  
function handleSlider (e) {  
  let value = e.target.value  
  root.style.setProperty('--slider', value)  
}
```

That was easy, huh?

Let me explain just in case I lost you

Let me explain just in case I lost you.

First off, make a reference to the slider element, `const range = document.querySelector('.booth-slider')`

Set up an event listener for when the value of the range input changes, `range.addEventListener('input', handleSlider)`

Write the callback, `handleSlider`

```
function handleSlider (e) {  
  let value = e.target.value  
  root.style.setProperty('--slider', value)  
}
```

The diagram shows the JavaScript code `root.style.setProperty('--slider', value)` with two annotations. A pink arrow points from the `root` variable to the text "Refers to HTML". Another pink arrow points from the `--slider` property to the text "Set the value INLINE". Below these annotations, the CSS style property is shown as `<html style="-- - slider: " "></html>`, with a pink underline under the `-- - slider:` part.

`root.style.setProperty('--slider', value)` says, get the `root` element (HTML), grab its style and set a property on it.

Handling the color changes

This is just as easy as handling the slider value change.

Here's how:

```
const inputs = document.querySelectorAll('.color-box > input')  
//as the value in the input changes, do something.  
  
inputs.forEach(input => {  
  input.addEventListener('input', handleInputChange)  
})  
  
function handleInputChange (e) {
```

```
let value = e.target.value
let inputId = e.target.parentNode.id

let inputBg = `--bg-${inputId}`
root.style.setProperty(inputBg, value)
}
```

Make a reference to all the text inputs, `const inputs = document.querySelectorAll('.color-box > input')`

Set up an event listener on all the inputs:

```
inputs.forEach(input => {
  input.addEventListener('input', handleInputChange)
})
```

Write the `handleInputChange` function:

```
function handleInputChange (e) {
  let value = e.target.value
  let inputId = e.target.parentNode.id
  let inputBg = `--bg-${inputId}`
  root.style.setProperty(inputBg, value)
}
```

```
function handleInputChange(e) {
  let value = e.target.value
  let inputId = e.target.parentNode.id
  let inputBg = `--bg-${inputId}`
  root.style.setProperty(inputBg, value)
}
```

The diagram illustrates the execution of the `handleInputChange` function with the following annotations:

- Get the value typed in the box**: Points to `e.target.value` in the first line.
- Get the ID of the input containing box**: Points to `e.target.parentNode.id` in the second line.
- Compose the variable**: Points to the template literal ``--bg-${inputId}`` in the third line.
- Set the variable inline**: Points to the `setProperty` call in the fourth line.

Phew...

That's it!

Project's done.