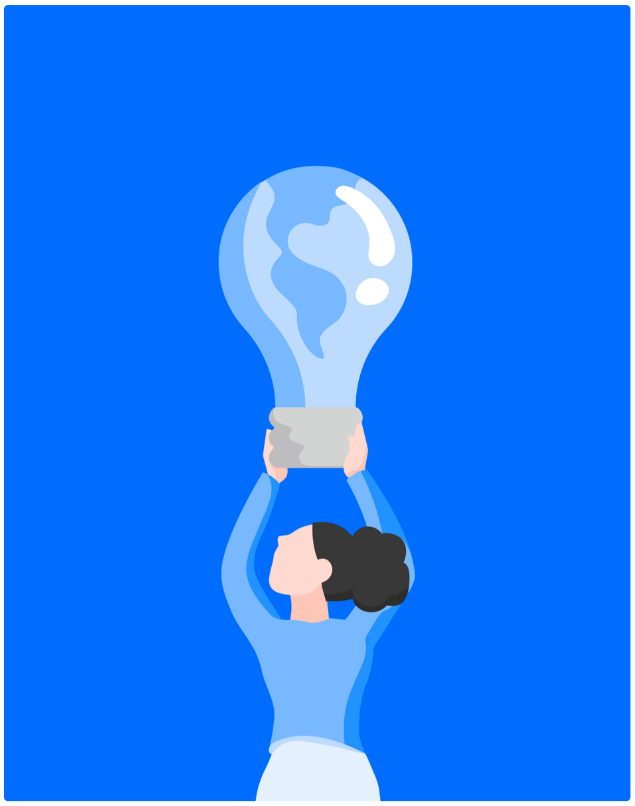
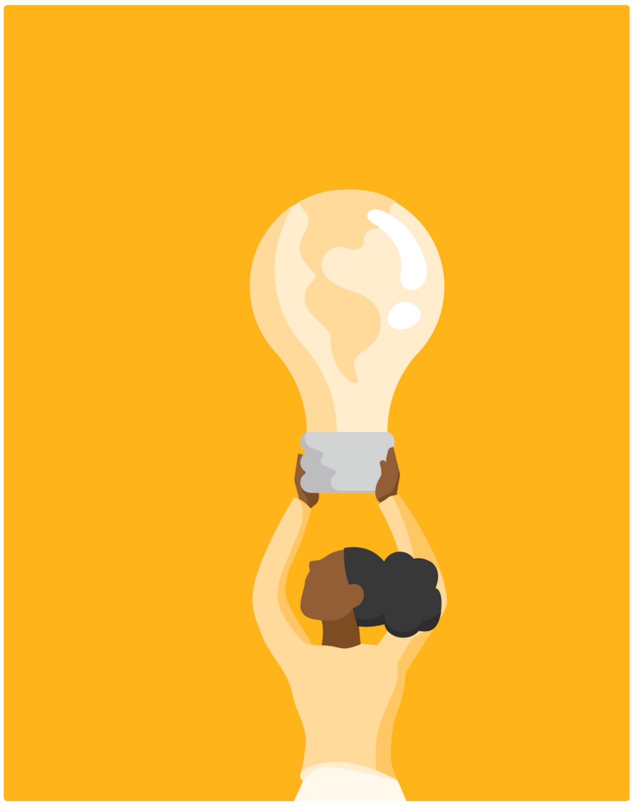
Pictogram Processing

Pictograms are vector graphics illustrations that are more complex than icons. They can include multiple color variations, changing background colors, accent colors, and skin colors if a person is in the illustration. These illustrations will use brand colors, but include multiple different shades of those brand colors.





You can view more examples here: https://drive.google.com/drive/folders/1-j3iOBGn3VIESog-LqarGlSDCTm94sMh - Connect your Google account

For Angelou icons, we automatically handle the colors of the icon by extracting the values into CSS variables. The various color changes across pictograms will make automatic variable extraction difficult--nearly impossible. Still, the Angelou integration will have to ensure that these illustrations allow for color customization.

Formatting

The pictograms will have to go through the SVGR process just like our icons. We should move our icons into a new `/assets/icons` folder, and create a new folder `/assets/pictograms`. We'll maintain our `/assets/index.ts` file and use that as a jumping point for our assets.

Naming

We don't have to do anything crazy. Similar to how we suffix our icons with SVG, I think we can suffix with `Picto` to differentiate the assets.

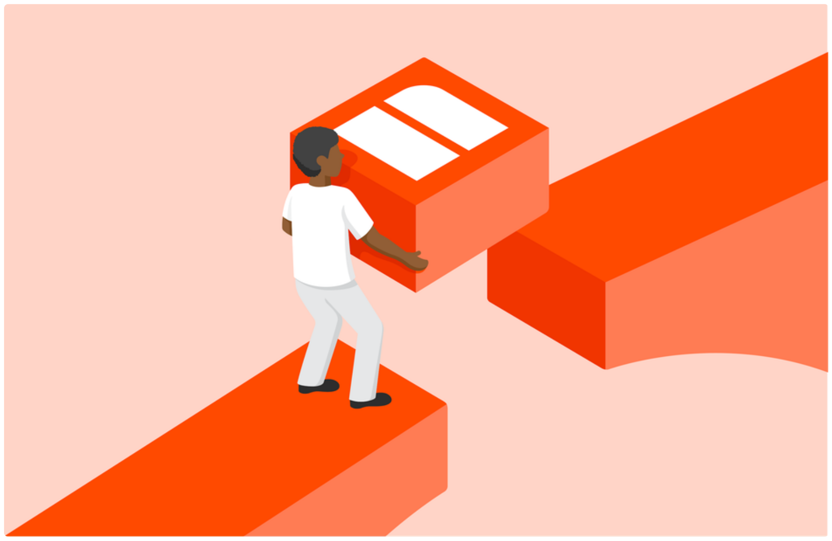
Accessibility

Since these are used as images, we should probably append an accessible description for each image. This will be done by using the Icon prop and specifying `title` and `isDecorative=false`. However, since our other components use `SvgComponent` as a prop, which expects a raw svg, we'll have to find another way to ensure the presence of the correct aria labels and title tags. This work was already done working on the Icon component, so we can copy that accessibility approach.

Colors & Control

So let's say each pictogram has 4 brand colors and 5 skin tones, that means each picto will have 20 variations. That's a little too much. While brand colors are more difficult to change, we *could try* to make some assumptions about skin color. If we make the skin color controllable via variables, we can reduce the amount of variations to *just* the number of brand colors. That'd be great.

We know the skin colors are about four main shades, but if you look closely in the pictogram above, skin may have different shading to add depth. So that expands the possible color palette, however it looks like only one additional shade is used so we may be able to make another assumption on the skin color shading.



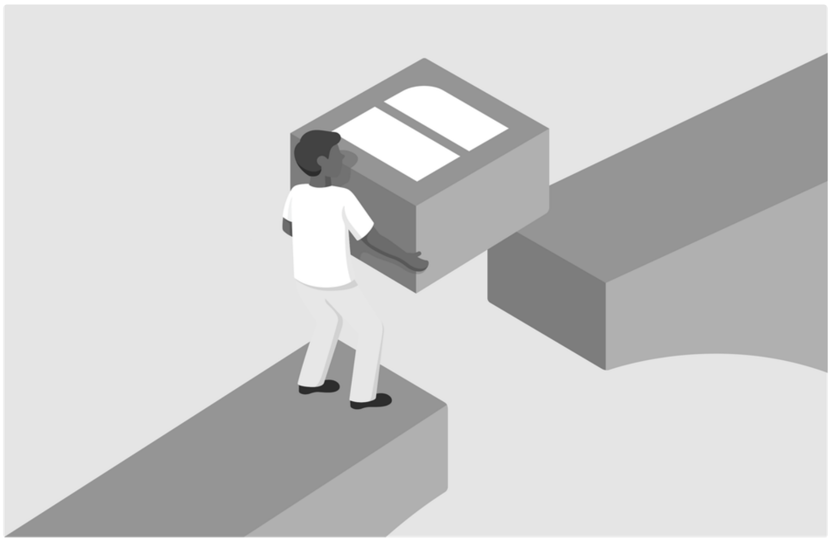
Here you can see one main shade and one darker shade to show shadows

SVGR

With this in mind, we can use similar methods to our past replacements where we can replace the skin colors with `var(--angelou-picto-skin-color-primary)` and `var(--angelou-picto-skin-color-secondary)` . That way we can create another helper function that will take care of creating these skin colors.

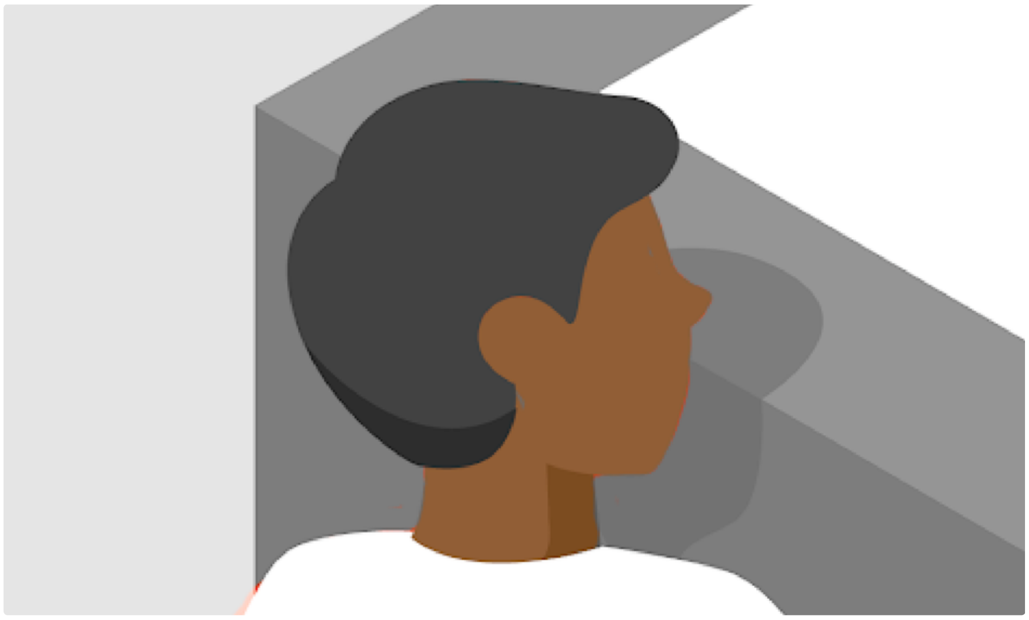
Percentage Saturation Approach

I have an idea that might make this generation really cool. Hypothetically, we can do a calculation about each path's color lightness compared to the darkest color, and generate the color variables based on that. I think it's easier to visualize this as a black and white image:


Without skin tone color

Isolate Skin Tones

First step would be to isolate our color palette of skin tones. Those should be “pulled out” so we know not to touch them. It could be possible to do this automatically, but the sure-fire way is to allow the user to “paint” out the colors, AKA click the paths in the SVG that are skin. We also have to differentiate between the primary skin tone and any shadows or highlights.


A closeup of the different shading a character can have

After painting out the skin tones, we end up with something like this:

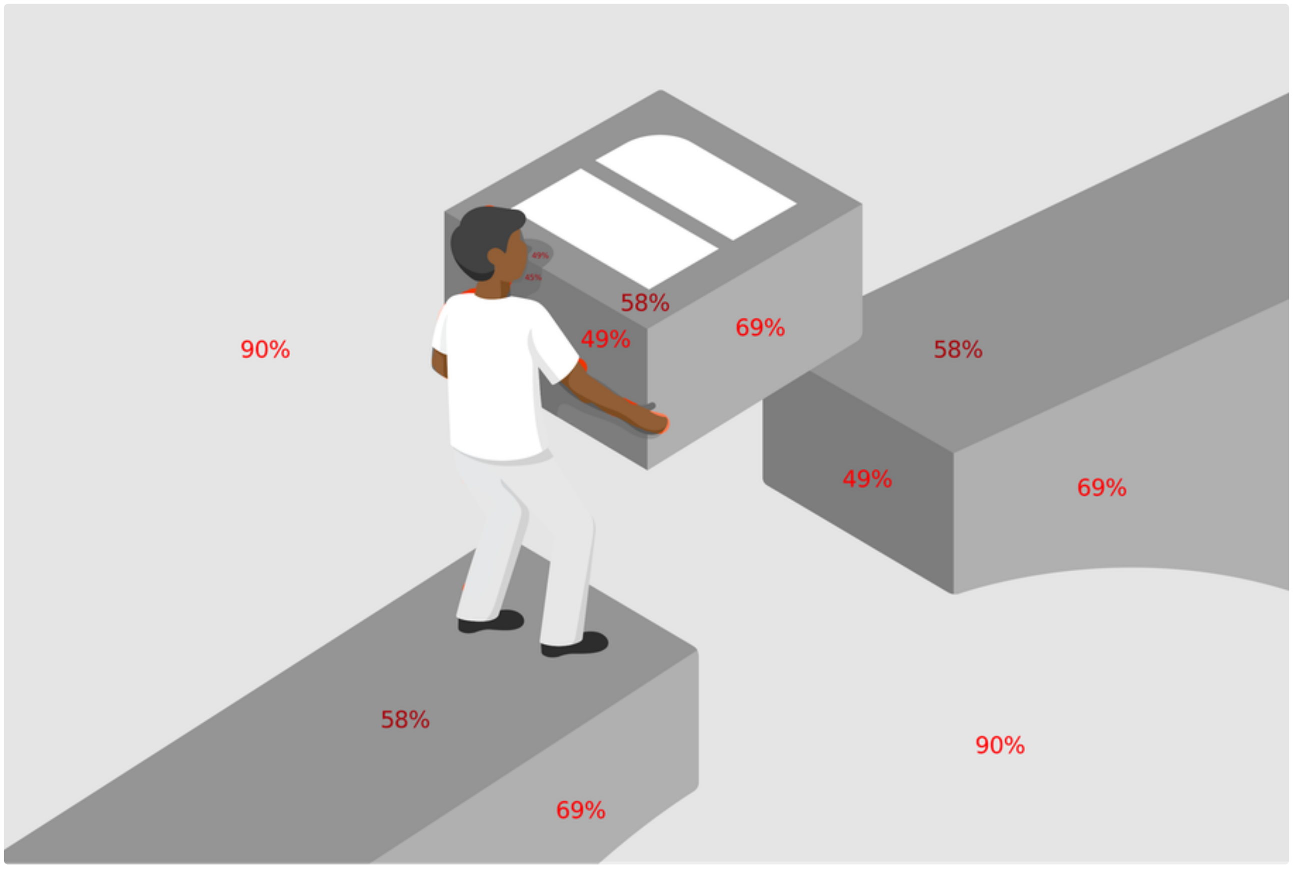
With skin tone color

Isolate Constant Colors

Sometimes our pictogram will include colors that should be ignored. Perhaps the hair should always be black, or a logo should always be blue. We'll want to allow users to paint out colors that will remain constant between brand color changes. After these paths are selected, we can remove them from any further processing.

Calculate Palette Lightness Across Paths

After this, we can figure out the lightness percentage for all remaining paths:



More specifically, these percentages are derived from selecting the existing fill or stroke color from the path, converting the HEX value to HSL, and pulling out the third value for lightness. In our replacement, the other two values will be discarded and replaced by our static brand color constant Hue and Saturation values.

Below is a table showing the Hue and Saturation values for each of our darkest brand colors. You can get this value by loading the HEX into a color picker and pulling out the H and S values from HSL.

Color	Hue Degrees	Saturation Percentage
Blue	215	100
Green	173	68
Yellow	36	78
Red	5	82

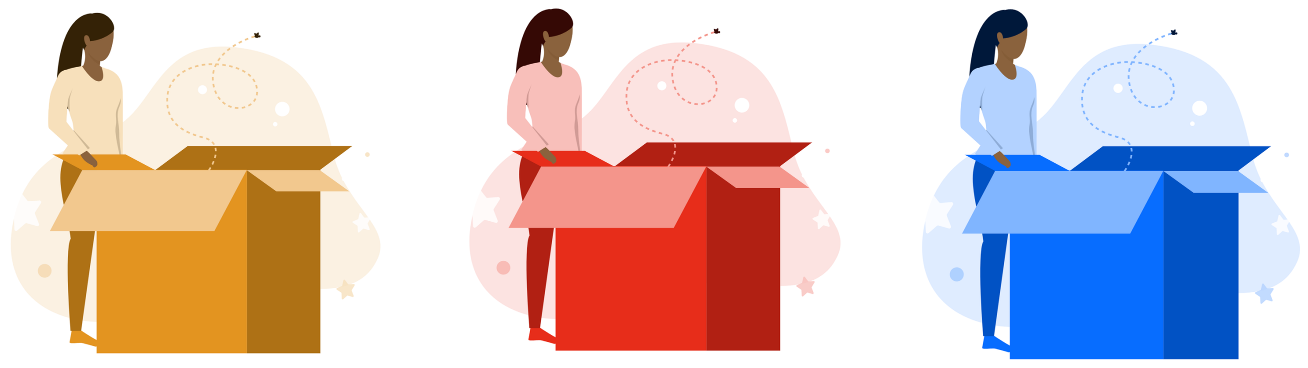
Knowing all of this, we can then loop through each path and replace the fill and/or stroke value with a CSS variable. So a fill will become `` `hsl(var(--active-h), var(--active-s), ${hsl[2]}%)`; ``

`--active-h`: the active brand color hue value

`--active-s`: the active brand color saturation value

`hsl[2]`: the lightness percentage value of the original color (exemplified by the graphic above)

Together, this will reproduce the graphic in any color we want, just by changing the `--active-s` and `--active-h` variables.

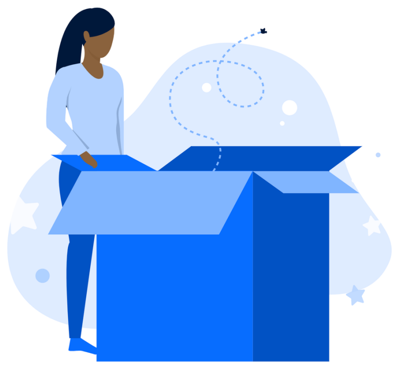


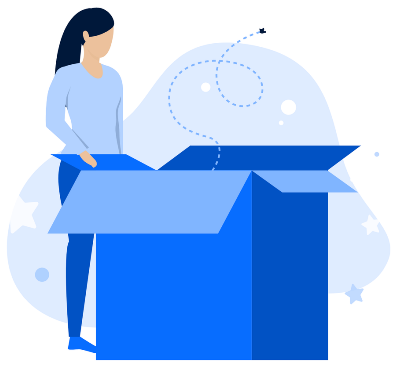
Handling Skin Tones

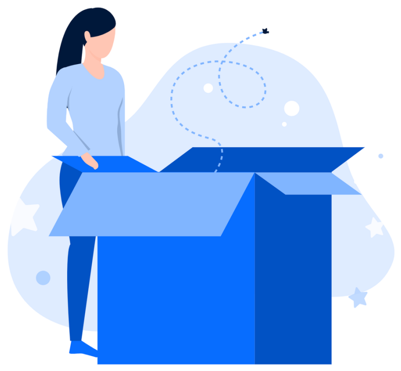
Changing skin tones works similarly. Since we isolated the paths beforehand and identified them as primary and shadow skin tones, we can use more CSS variable constants to save our different skin tones.

Skin Tone	Primary Variable	Secondary Variable	Primary	Shadow
Fair	`--skin-tone-fair-primary`	`--skin-tone-fair-shadow`	`#ffc6b4`	`#ffab90`
Midtone	`--skin-tone-midtone-primary`	`--skin-tone-midtone-shadow`	`#ecc19c`	`#dfb38b`
Dark	`--skin-tone-dark-primary`	`--skin-tone-dark-shadow`	`#8a613e`	`#764f2a`

And now we can replace those static fills and strokes with a CSS variable: `--active-skin-tone-primary` and `--active-skin-tone-shadow`. Then we can specify which skin tone is active, and voila!


Dark


Midtone


Fair

CSS Variable Architecture

When designing this, I wanted to prioritize two things:

1. Completely customizable by consumers
2. Usable by designers via copy/paste

The best way to approach this is to use CSS variables. We can even tier our CSS variables to allow internal and external control. What does this mean?

Internal Control

After processing, we want the SVG to work right out of the box. We don't want to *require* that consumers configure a variable architecture. We do this by appending CSS variable rules to the root `<svg>` element. These variables are referenced throughout the internal paths of the SVG. So let's say our root will have variables `active-h`, `active-s`, `active-skin-tone-primary`, and `active-skin-tone-shadow`. Our paths inside the SVG *only* reference these top-level variables. By setting these variables to values, we'll always have a functioning SVG even if the consumer doesn't customize the colors.

External Control

However, we want to follow our two parameters and allow customization. So what we can do is set the SVG's properties to consumer an external variable if it exists, and set a fallback variable if it doesn't. So inside our SVG tag's properties, we set it like this:

```
--active-skin-tone-primary: var(--angelou-active-skin-tone-primary, #8a613e);
```

With this approach, consumers can externally set the variable `--angelou-active-skin-tone-primary` to be whatever they want, and the SVG will know to use that. If it isn't present however, we can use the fallback HEX code provided.

Within the app I made (link below), we update that fallback HEX color to be whichever value is currently selected. So if the dropdowns specify blue brand color with dark skin tone, we update the fallback variables to be just that, and the outputted code is updated. This allows designers to copy/paste the SVG markup and paste it into Figma, or engineers to copy/paste one-time-needed code, allowing for easily customizable SVGs.

Implementation Into Workflow

Luckily, implementation into Angelou won’t be too difficult. Work is already underway to update our processing of SVG files to handle pictograms. The only question is preparation.

Here is how I think it should happen:

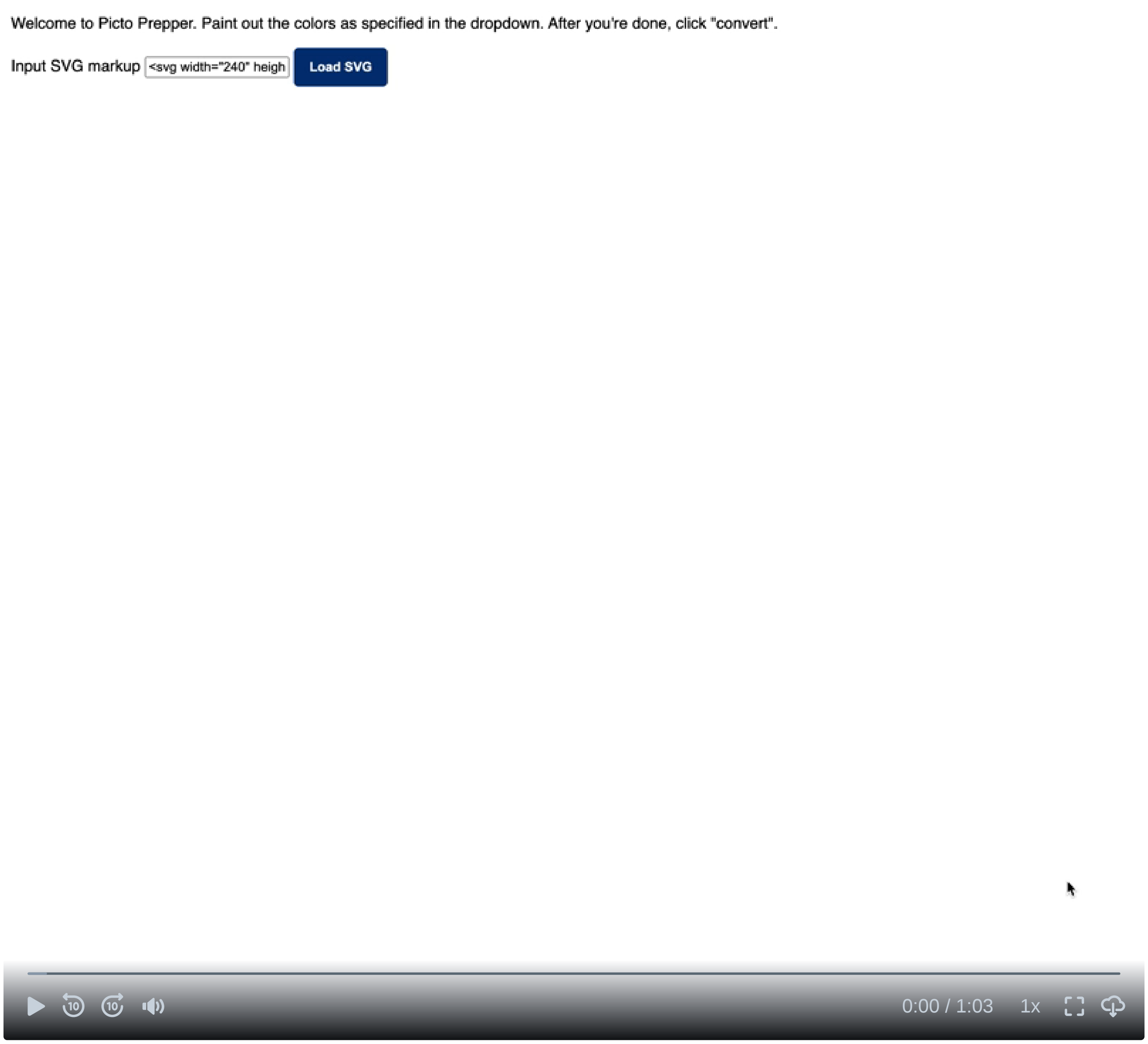
1. (One time) Designers outline skin-tone constants:
    a.
    ```
    --skin-tone-fair-primary: #ffc6b4;
    --skin-tone-fair-shadow: #ffab90;
    --skin-tone-midtone-primary: #ecc19c;
    --skin-tone-midtone-shadow: #dfb38b;
    --skin-tone-dark-primary: #8a613e;
    --skin-tone-dark-shadow: #764f2a;
    ```
    b. These values will live in Angelou and do not have to be frequently changed
2. Designers process the pictograms in the program before handing off to engineers
    a. Different colors are painted and variables are settled (automatically) beforehand
3. Engineers save the processed pictogram as an SVG in Angelou
4. (One time) Engineers create pattern for consumers to override skin & brand colors

Conclusion

Now we can output this SVG, use it in our build, and can alter any skin tones / backgrounds without hosting multiple versions of the same graphic.

To make everything easier, I created a tool that does all of this for you. Here is the URL: ● Pictogram Prepper 6000

and here is a video of it in action:



Welcome to Picto Prepper. Paint out the colors as specified in the dropdown. After you're done, click "convert".

Input SVG markup <svg width="240" heigh