Below is one way to modify the AdaptiveOklab JavaScript library so that when you choose among three surround conditions (say, "white", "gray", and "dark")—each with its own exponent—the hue (i.e. the angle in the a–b plane) is preserved. In this proposal, we let the nonlinearity exponent be different for each surround but then apply a correction factor (only to the opponent channels a and b) to “undo” the shift in scaling relative to the standard (gray) case. In other words, if the standard gray surround uses an exponent of 1⁄3, we compute a correction factor that compensates for using a different exponent for white or dark surrounds.

Below is an updated version of the library code:

```
// adaptiveOklab.js

export class AdaptiveOklab {
  /**
   * Creates an instance of AdaptiveOklab.
   *
   * @param {Object} options - Options for surround conditions.
   * @param {string} options.surround - One of "white", "gray", or "dark".
Default is "gray".
   * @param {number} [options.x0=0.5] - A representative cone response value
used for computing a correction factor.
   */
  constructor(options = {}) {
    // Surround can be "white", "gray", or "dark"
    this.surround = options.surround || 'gray';

    // Define the exponents for different surround conditions.
    // These are example values.
    this.surroundExponents = {
      white: 0.35, // for a bright (white) surround
      gray:  1 / 3, // standard dark-to-gray surround (0.33333…)
      dark:  0.31  // for a darker surround
    };

    // Representative LMS value for computing a correction factor.
    // This value should be chosen to represent typical cone responses.
    this.x0 = options.x0 || 0.5;

    // Predefined matrices for conversion.
    // Matrix for converting from XYZ to LMS.
```

```
    this.M_XYZ_to_LMS = [
      [ 0.8189330101,  0.3618667424, -0.1288597137],
      [ 0.0329845436,  0.9293118715,  0.0361456387],
      [ 0.0482003018,  0.2643662691,  0.6338517070]
    ];

    // Matrix for converting from nonlinear LMS to Oklab.
    // (This was optimized for the standard 1/3 exponent.)
    this.M_LMS_to_Oklab = [
      [ 0.2104542553,  0.7936177850, -0.0040720468],
      [ 1.9779984951, -2.4285922050,  0.4505937099],
      [ 0.0259040371,  0.7827717662, -0.8086757660]
    ];
  }

  /**
   * Returns the exponent corresponding to the current surround condition.
   *
   * @returns {number} Exponent value.
   */
  getExponent() {
    return this.surroundExponents[this.surround] || (1 / 3);
  }

  /**
   * Computes a sign-preserving power-law nonlinearity.
   *
   * @param {number} value - The input value.
   * @param {number} exponent - The exponent to use.
   * @returns {number} The transformed value.
   */
  signPreservingPow(value, exponent) {
    return Math.sign(value) * Math.pow(Math.abs(value), exponent);
  }

  /**
   * Multiplies a 3x3 matrix with a 3x1 vector.
   *
   * @param {Array<Array<number>>} matrix - The 3x3 matrix.
   * @param {Array<number>} vector - The 3-element vector.
   * @returns {Array<number>} The resulting vector.
   */
  multiplyMatrixVector(matrix, vector) {
    return matrix.map(row =>
      row.reduce((acc, cur, i) => acc + cur * vector[i], 0)
    );
```

```
  }

  /**
   * Computes a correction factor for the opponent channels.
   *
   * The idea is that the original Oklab used an exponent of 1/3.
   * When a different exponent is used, the nonlinear compression scales the
LMS
   * responses differently. To preserve hue uniformity (i.e. the a–b angle),
   * we apply a correction factor computed at a representative cone response
value x0.
   *
   * @param {number} p - The new exponent.
   * @returns {number} Correction factor.
   */
  getCorrectionFactor(p) {
    // The desired behavior is that for the standard exponent (1/3) the
factor is 1.
    // For other exponents, we approximate a uniform correction using a
representative value x0.
    // Correction factor = x0^( (1/3) - p )
    return Math.pow(this.x0, (1 / 3) - p);
  }

  /**
   * Converts from XYZ to adaptive Oklab.
   *
   * @param {number} X - X component of XYZ.
   * @param {number} Y - Y component of XYZ.
   * @param {number} Z - Z component of XYZ.
   * @returns {Object} Object with L, a, and b.
   */
  fromXYZ(X, Y, Z) {
    // 1. Convert from XYZ to LMS.
    const LMS = this.multiplyMatrixVector(this.M_XYZ_to_LMS, [X, Y, Z]);

    // 2. Apply the adaptive exponent to each LMS channel.
    const p = this.getExponent();
    const LMSprime = LMS.map(channel => this.signPreservingPow(channel, p));

    // 3. Convert from nonlinear LMS to Oklab.
    let lab = this.multiplyMatrixVector(this.M_LMS_to_Oklab, LMSprime);

    // 4. Apply a correction to the opponent channels to preserve hue
uniformity.
    //    (This correction approximates the inverse effect of changing the
```

```
exponent.)
    const correction = this.getCorrectionFactor(p);
    // Adjust only the a and b channels (the opponent channels),
    // leaving L unchanged.
    lab[1] *= correction;
    lab[2] *= correction;

    return { L: lab[0], a: lab[1], b: lab[2] };
  }

  /**
   * Converts from sRGB to adaptive Oklab.
   * sRGB is assumed to be in the [0, 1] range.
   *
   * @param {number} r - Red channel.
   * @param {number} g - Green channel.
   * @param {number} b - Blue channel.
   * @returns {Object} Object with L, a, and b.
   */
  fromSRGB(r, g, b) {
    // Convert sRGB to linear sRGB.
    const linearize = (v) =>
      v <= 0.04045 ? v / 12.92 : Math.pow((v + 0.055) / 1.055, 2.4);
    const rLin = linearize(r);
    const gLin = linearize(g);
    const bLin = linearize(b);

    // Convert linear sRGB to XYZ using the standard matrix.
    const X = 0.4124564 * rLin + 0.3575761 * gLin + 0.1804375 * bLin;
    const Y = 0.2126729 * rLin + 0.7151522 * gLin + 0.0721750 * bLin;
    const Z = 0.0193339 * rLin + 0.1191920 * gLin + 0.9503041 * bLin;

    return this.fromXYZ(X, Y, Z);
  }
}
```

Explanation

1. **Surround-Dependent Exponents:**
   The constructor now accepts a `surround` option (with allowed values `"white"`, `"gray"`, or `"dark"`). A lookup table (`this.surroundExponents`) assigns an exponent value for each condition. For instance, the standard gray surround uses 1⁄3 (0.33333…), while you

might choose a slightly higher exponent for a white surround (e.g. 0.35) and a lower one for a dark surround (e.g. 0.31).

2. **Uniform Application of the Nonlinearity:**
   The method `fromXYZ` applies the same exponent to all three LMS channels via the `signPreservingPow` function. This uniformity is what preserves the hue—because all channels are scaled in the same way, the ratios (and therefore the angles) between channels remain the same.
3. **Correction Factor for Opponent Channels:**
   Because the matrices in Oklab were optimized for the 1⁄3 exponent, changing the exponent alters the effective scaling of the nonlinearly compressed LMS values. To compensate, a correction factor is computed (using a representative value `x0`) and applied to only the a and b channels (the opponent channels). This helps preserve the appearance of a given hue (for example, 260°) across different surrounds, even though the lightness and chroma might shift slightly.
4. **Example Usage:**
   You can now create an instance for a particular surround condition:

```
import { AdaptiveOklab } from './adaptiveOklab.js';

// Use the "white" surround exponent.
const aok = new AdaptiveOklab({ surround: 'white' });
const lab = aok.fromSRGB(0.2, 0.5, 0.8);
console.log(`Adaptive Oklab (white surround): L=${lab.L}, a=${lab.a},
b=${lab.b}`);
```

This design provides a starting point for an adaptive Oklab model with three preset surround conditions. While the correction factor is an approximation (based on a representative value), it ensures that—for a given surround choice—the hue (i.e. the angle in the a–b plane) remains consistent across lightness and chroma variations.