包括的な調査：すべてのLUTが同一のカラー効果を適用する理由

I. エグゼクティブサマリー

本報告書は、Next.jsアプリケーションにおいて、複数の異なる3D LUTファイルがロードされているにもかかわらず、画像に同一のカラー変換効果が適用される問題の根本原因を調査したものである。主要な調査結果は、WebGLのテクスチャユニット管理とユニフォーム変数割り当てにおける不正確さが問題の根源にあることを示している。これにより、複数のsampler2Dユニフォームが同じ基盤となるWebGLTextureを参照してしまう状況が発生している。これに関連して、3Dから2Dへのテクスチャ座標マッピングや、複数のLUTを組み合わせるロジックにも問題が存在する可能性がある。

特定された根本原因は、主にシェーダーユニフォームの競合であると考えられる。これは、複数のsampler2Dユニフォーム（u_lut1、u_lut2など）が、JavaScriptコード内で意図せず同じテクスチャユニットインデックスを指すように設定されている場合に発生する。また、テクスチャの上書きも考えられる。これは、gl.texImage2Dが単一のWebGLTextureオブジェクトまたは単一のテクスチャユニットに対して繰り返し呼び出され、適切なユニークなオブジェクトの作成とバインディングが行われていない場合に発生する。

解決策としては、各LUTを独自のWebGLTextureオブジェクトにロードし、個別のテクスチャユニットにバインドし、フラグメントシェーダー内の対応するsampler2Dユニフォームによって正しく参照されるようにすることが挙げられる。Canvas2Dフォールバックの場合には、JavaScriptのピクセル操作ロジックが異なるLUTデータに正しくアクセスし、適用するように修正する必要がある。デバッグには、Chrome DevToolsとSpector.jsを活用し、GPUの状態、テクスチャの内容、ユニフォームの値を検査するとともに、データフロー検証のための戦略的なconsole.logステートメントを使用する。

II. はじめに

問題提起

Next.jsアプリケーションにおける画像処理において、重要な異常が確認されている。複数の.cube形式の3D LUTファイル（例：Anderson.cube、Blue sierra.cube）が正常にロードされ解析されているにもかかわらず、すべての適

用されたLUTが画像に対して視覚的に同一のカラー変換効果をもたらしている。ログは各LUTの異なるデータを示しており、コンソールエラーも報告されていないため、問題は画像処理およびカラー変換パイプライン、特にWebGL2またはCanvas2Dのレンダリング段階にあることが示唆されている。

調査目的

本調査の目的は、この均一性の根本原因を正確に特定し、WebGL2とCanvas2Dのフォールバックメカニズムの両方に対応する堅牢な解決策を提案し、効果的なデバッグ手法を概説し、明確で実行可能な実装ロードマップを提供することである。

技術的背景

アプリケーションはNext.jsとTypeScriptで構築されており、高性能な画像処理のためにWebGL2を利用し、WebGL2が利用できない場合やサポートされていない場合にはCanvas2Dにフォールバックする。LUTは64^3解像度の.cubeファイルであり、GPUアップロードのために2Dテクスチャ表現（例：4096x64）に変換される。フラグメントシェーダーは、3D LUT変換を三線形補間を用いて適用する役割を担っている。

III. 技術詳細：3D LUTの適用を理解する

A. 3D LUTの基礎

3Dルックアップテーブル（LUT）は、入力カラー値（RGB）を出力カラー値に変換するためのデータ構造である。これらは本質的に3Dカラー空間のグリッド（「キューブ」）であり、各点（または「ノード」）には変換されたカラーが含まれている。中間色は補間によって導き出される [1]。

.cubeファイルはこれらのグリッド点を指定し、通常は赤、緑、青の各チャンネルに対して定義される。64^3のLUTは、各軸に64のサンプルがあることを意味し、結果として64 * 64 * 64 = 262,144のデータ点となる (ユーザーのクエリ)。LUTは、画像のコンテンツのコンテキストなしに入力カラーに基づいて直接ルックアップを実行するため、「ダム」である。例えば、グレインやブルームを直接適用することはできない。これらは主にカラーとコントラストの変換に使用される [1]。

B. WebGLにおける3D LUTのテクスチャ表現

WebGL2は3Dテクスチャ（gl.TEXTURE_3D）をサポートしており、3D LUTには理想的である。しかし、WebGL1（およびWebGL2が明示的に使用されていない場合のフォールバック）は、3Dデータを2Dテクスチャにマッピングする必要がある[5]。

64^3のLUTの場合、一般的なアプローチは、それを2Dテクスチャに「アンラップ」することである。例えば、64の「スライス」（2Dレイヤー）を横に並べて配置することで、4096x64のテクスチャ（幅が64 * 64 = 4096ピクセル、高さが64ピクセル）を作成する[2]。

.cubeファイルから解析されたFloat32Array形式のLUTデータは、gl.texImage2Dに適した形式、通常はUint8Array（UNSIGNED_BYTE形式の場合）に変換する必要がある。この場合、各カラーコンポーネントは0から255までの整数として表現される[7]。

C. WebGLにおけるLUTのレンダリングパイプライン

頂点シェーダーの役割

頂点シェーダーは主にジオメトリを設定し、テクスチャ座標（例：v_texcoord）をフラグメントシェーダーに渡す役割を担う。フルスクリーン画像処理の場合、これは多くの場合、画面全体を覆う単純なクワッドをレンダリングすることを含む[9]。

フラグメントシェーダーの applyLUT 関数

フラグメントシェーダーは、入力ピクセルのカラーを受け取る。その後、この入力カラーから3D座標（R、G、B）を計算し、の範囲に正規化して、LUTテクスチャをサンプリングする。この3D座標は、アンラップされた2D LUTテクスチャ上の2D (u,v)座標にマッピングされなければならない。例えば、64^3のLUTを4096x64にアンラップする場合、計算は通常以下のようになる：

- u座標はRおよびBチャンネルに依存する（例：(B_slice_index + R_normalized) / total_slices_along_u）。
- v座標はGチャンネルに依存する（例：G_normalized）。

これらの座標計算は、アンラップされたテクスチャの正確なレイアウトに細心の注意を払って実装する必要がある[8]。

三線形補間: スムーズなカラー遷移を得るためには、三線形補間が適用される。3D LUTを表現するために2Dテクスチャを使用する場合、これは2つの隣接する「スライス」（2Dレイヤー）に対して双線形補間を実行し、その後、3番目のカラーコンポーネント（3Dキューブの「深さ」または「スライス」を決定する）に基づいて、これら2つの双線形結果間で線形補間を行うことを意味する。三線形という用語は多義的である場合があるが、3Dテクスチャの場合、GL_LINEARフィルタリングがこれを処理する。3Dデータを表現する2Dテクスチャの場合、シェーダー内で手動で実装する必要がある[3]。

Sampler2Dユニフォームとテクスチャユニット

GLSLでは、sampler2Dユニフォームがテクスチャにアクセスするために使用される[9]。JavaScriptでは、

gl.getUniformLocationがこれらのユニフォームのロケーションを取得し、gl.uniform1iがそれらにテクスチャユニットインデックスを割り当てる[7]。

gl.activeTexture(gl.TEXTURE0 + unitIndex)は特定のテクスチャユニットをアクティブにし、gl.bindTexture(gl.TEXTURE_2D, textureObject)はWebGLTextureオブジェクトをそのアクティブなユニットにバインドする[7]。WebGLは少なくとも8つのテクスチャユニット（

gl.MAX_COMBINED_TEXTURE_IMAGE_UNITS）をサポートすることを保証しており、通常、フラグメントシェーダーには8つ、頂点シェーダーには4つが割り当てられる[22]。

IV. 調査結果と分析

A. WebGLテクスチャの作成と管理 (src/lib/webgl-utils.ts, src/lib/lutProcessor.ts)

テクスチャオブジェクトのライフサイクル

gl.createTexture()は、新しいWebGLTextureオブジェクトを作成するために使用される関数である。この関数が呼

び出されるたびに、一意のオブジェクト参照が返されるべきである[7]。

src/lib/webgl-utils.ts内のcreate3DLUTTextureが各異なるLUTファイル（Anderson.cube、Blue sierra.cubeなど）に対して呼び出される場合、それぞれに対して一意のWebGLTextureオブジェクトが生成されるはずである。問題は、この関数が単一のWebGLTextureオブジェクトを複数のLUTデータアップロードで誤って再利用または上書きしている場合、またはsrc/lib/lutProcessor.ts内のlutTextureMapがこれらのユニークなオブジェクトを正しく保存および取得していない場合に発生する。

テクスチャオブジェクトの再利用/上書きの可能性: ユーザーが「LUTは正常にロードされる」と「ログには異なるデータが表示される」と述べていることは、.cubeファイルの解析が異なるデータ配列（例：Float32Array）に正しく機能していることを示唆している。次のステップはcreate3DLUTTextureとgl.texImage2Dである。create3DLUTTextureが各LUTに対して呼び出される場合、新しいWebGLTextureオブジェクトを作成するはずである[18]。しかし、アプリケーションが

同じWebGLTextureオブジェクト（または同じテクスチャユニット）を繰り返しバインドし、異なるデータでgl.texImage2Dを呼び出す場合、最後にアップロードされたデータのみがそのテクスチャオブジェクトに保持される[28]。これは、すべてのLUTがGPU上で同じテクスチャデータを共有することにつながり、視覚的に同一の効果をもたらす直接的な原因となる。この状況は、ロードされた最後のLUTの効果、またはテクスチャが適切に更新されない場合の単一のデフォルトLUTの効果が、すべての画像に表示されるという形で現れる可能性がある。

データアップロードの検証

gl.texImage2Dは、ピクセルデータをGPUテクスチャにアップロードするために使用される。この関数は、internalFormat、width、height、srcFormat、srcType、およびデータ配列（例：Uint8Array）などのパラメータを受け取る[7]。

Float32ArrayからUint8Arrayへの変換、およびgl.texImage2Dが各LUTに対して異なる、正しいUint8Arrayで呼び出されていることを確認する必要がある。「異なるデータ」を示すログは重要だが、最終的なチェックはGPU上で行われるべきである。

データ変換またはアップロードの不整合の可能性: 「すべてのLUTがログには異なるデータを示すが、視覚的には同一の効果を示す」という事実は、CPU側のJavaScriptからデータが正しく出力されていることを強く示唆している。次の失敗点は、gl.texImage2Dを介したGPUへの転送である。もしFloat32Array（解析されたデータ）からUint8Array（GPU対応データ）への変換に欠陥がある場合、またはtexImage2D呼び出し中に同じUint8Arrayバッファが誤って再利用または参照されている場合、GPUは同一のデータを受け取ることになる。これは、create3DLUTTextureがUint8Arrayデータを生成する際に、微妙なバグにより常に同じ値でデータを埋めてしまう場合、またはコピーではなく参照が誤って渡されている場合に発生する可能性がある。このような状況は、正しいバインディングが行われていても、GPUが同一のテクスチャデータを持つことになり、視覚的に同一の効果をもたらす。

テクスチャ参照の共有 (lutDataMap/lutTextureMap)

これらのマップは、ロードされたLUTを管理するために非常に重要である。lutTextureMapは、LUT名またはIDをキーとして、一意のWebGLTextureオブジェクトを格納する必要がある。

create3DLUTTexture関数の実行をトレースする必要がある。この関数は常に新しいWebGLTextureオブジェクトを作成しているのか、それともすでに存在する場合はキャッシュ/マップから取得しているのか。もし新しいオブジェクトを作成する意図がある場合、gl.createTexture()の呼び出しがスキップされていないか、または以前に作成されたテクスチャが新しいLUTに対して誤って再利用されていないかを確認する必要がある。もしキャッシュする意図がある場合、キャッシュキーが各異なるLUTデータに対して真に一意であることを確認する必要がある。

マップの不整合または不正確なキャッシュロジックの可能性: lutDataMapとlutTextureMapの存在は、ユニークなLUTデータとそれに対応するGPUテクスチャを管理する意図があることを示唆している。もしlutTextureMapが単純なシングルトンとして実装されている場合、またはテクスチャを保存/取得するために使用されるキーが、LUTファイルコンテンツごとに真に一意ではない（例：上書きされる汎用識別子に基づいている）場合、システムは異なるLUTに対して同じWebGLTextureオブジェクトを取得する可能性がある。あるいは、create3DLUTTextureが複数回呼び出されるが、gl.bindTextureが常に同じテクスチャユニットを指し、そのユニットに新しく作成された一意のテクスチャオブジェクトが最初にバインドされていない場合、texImage2Dの呼び出しは、以前にそのユニットにバインドされていたテクスチャオブジェクト上のデータを上書きしてしまう。このような状況は、たとえ異なるテクスチャユニットを指していても、すべてのu_lutユニフォームが最終的に同じ基盤となるテクスチャデータからサンプリングしてしまうというシナリオにつながる可能性がある。

LUTファイル名	解析直後の生データサンプル (CPU)	texImage2D準備後のデータサンプル (CPU)	読み戻しデータサンプル (GPU, Spector.js)	Spector.js テクスチャプレビュー (視覚確認)
Anderson.cube	``	[r1, g1, b1, a1,...]	[r1, g1, b1, a1,...]	(Anderson.cubeのテクスチャ画像)
Blue sierra.cube	``	[r2, g2, b2, a2,...]	[r2, g2, b2, a2,...]	(Blue sierra.cubeのテクスチャ画像)
F-PRO400H.cube	``	[r3, g3, b3, a3,...]	[r3, g3, b3, a3,...]	(F-PRO400H.cubeのテクスチャ画像)

表：LUTデータフロー検証ポイント
この表は、ファイル解析からGPUへのデータ転送までのデータ整合性を段階的に検証することを可能にする。もし「解析直後の生データサンプル」が異なっているにもかかわらず、「texImage2D準備後のデータサンプル」や「読み戻しデータサンプル」が複数のLUT間で同一である場合、それはCPU上またはGPUアップロード中のデータ変換、コピー、または上書きの問題を示している。

B. WebGLシェーダーパイプライン分析 (フラグメントシェーダー、テクスチャバインディングロジック)

テクスチャユニットバインディングの検証

WebGLは「テクスチャユニット」（例：TEXTURE0、TEXTURE1、TEXTURE2など）をテクスチャへの参照の配列として使用する。シェーダー内の各sampler2Dユニフォームは、gl.uniform1iを介してどのテクスチャユニットを使用するかを明示的に指示される [7]。

問題の核心はここにある可能性が高い。もしシェーダー内でu_lut1、u_lut2、u_lut3が宣言されている場合、それらはgl.uniform1iを使用して異なるテクスチャユニットインデックス（例：それぞれ0、1、2）に割り当てられなければならない。同時に、対応するWebGLTextureオブジェクト（異なるLUTデータを含む）は、gl.activeTextureとgl.bindTextureを使用して、これらの同じ異なるテクスチャユニットにバインドされなければならない。

主要な候補：シェーダーユニフォームの競合: 複数の異なるLUTがロードされているにもかかわらず、視覚的に同一の効果が得られる最も直接的な原因は、シェーダーが常に同じテクスチャからサンプリングしていることである。これは、GLSLフラグメントシェーダー内の複数のsampler2Dユニフォーム（例：uniform sampler2D u_lut1; uniform sampler2D u_lut2;）が、JavaScriptコード内で同じテクスチャユニットインデックス（例：gl.uniform1i(u_lut1Location, 0); gl.uniform1i(u_lut2Location, 0);）に割り当てられている場合に発生する。たとえgl.activeTexture(gl.TEXTURE1); gl.bindTexture(gl.TEXTURE_2D, differentLUTTexture);が呼び出されても、シェーダーがすべてのLUTに対してTEXTURE0からサンプリングするように指示されている場合、TEXTURE1、TEXTURE2などに存在する異なるテクスチャは単純に無視される。これは、WebGL開発における一般的な落とし穴であり、JavaScriptの状態（アクティブなテクスチャユニット、バインドされたテクスチャ）とシェーダーのユニフォーム値（どのテクスチャユニットからサンプリングするか）が同期されなかったり、誤って設定されたりすることによって発生する。この状況は、ログ内の異なるLUTデータが視覚的に同一の結果をもたらす理由を直接説明する。シェーダーは常にGPU上の特定のLUTテクスチャのみを参照しているためである。

ユニフォーム変数管理

gl.getUniformLocationはユニフォームへのハンドルを取得するために使用される。サンプラーの配列（例：uniform sampler2D u_luts;）の場合、getUniformLocationはu_luts、u_lutsなど、または配列全体を設定する場合はベース名u_lutsに対して呼び出すことができる [11]。

gl.getUniformLocationがu_lut1、u_lut2、u_lut3に対して一意のロケーションを正しく取得していることを確認する必要がある。もしこれらが配列ユニフォーム（u_luts[N]）の一部である場合、GLSLでのインデックス付け（動的な場合）がWebGL1/2でサポートされていることを確認する必要がある。サンプラー配列の動的なインデックス付けには制限があるためである [17]。

サンプラー配列の動的インデックス付けの制限: フラグメントシェーダーがsampler2Dユニフォームの配列（例：uniform sampler2D u_luts[N];）を使用しようとし、非定数インデックス（例：texture2D(u_luts[i],...)でiがvaryingまたは計算された値である場合）でアクセスする場合、WebGL1（および一部のWebGL2コンテキスト）は、この「動的インデックス付け」をサポートしない可能性がある [17]。このような場合、シェーダーは

u_lutsからサンプリングをデフォルトにするか、未定義の動作を示す可能性があり、事実上すべてのLUT適用が同じ最初のLUTを使用することになる。シェーダーがエラーなしでコンパイルおよびリンクされているように見えても、この実行時の動作が観察される均一性につながる可能性がある。これは、解決策として、個別のsampler2Dユニフォーム（u_lut1、u_lut2など）を使用し、それらを個別に設定するか、配列アプローチを希望する場合はGLSLバージョンとコンテキストが動的インデックス付けをサポートしていることを確認する必要があることを意味する。

フラグメントシェーダーの applyLUT ロジック (3D→2D座標変換と三線形補間)

入力vec3（正規化されたRGB）からアンラップされた2D LUTテクスチャのvec2（テクスチャUV）への変換は非常に重要である。64^3のLUTを4094x64にマッピングする場合、計算は通常次のようになる：

- float sliceSize = 1.0 / 64.0;
- float sliceOffset = 0.5 / (4096.0); // Half texel offset
- float x = inputColor.r * (63.0/64.0) + (floor(inputColor.b * 63.0) * sliceSize) + sliceOffset; (近似値であり、RとBを水平軸に注意深くマッピングする必要がある)
- float y = inputColor.g;

[3]

三線形補間が3DテクスチャのGL_LINEARによって自動的に処理されない場合、8つの周辺テクセルをサンプリングしてブレンドすることにより、シェーダー内で手動で実装する必要がある。LUTが2Dにアンラップされている場合、これは2つの隣接する「スライス」からサンプリングして補間することを伴う [3]。

座標計算エラーの可能性: applyLUT関数内の3D -> 2D座標マッピングに欠陥がある場合、実際のLUTデータに関係なく、すべての入力カラーがLUTテクスチャの同じ領域にマッピングされる可能性がある。例えば、座標計算の青色チャンネルコンポーネントが常にゼロである場合、またはスライス計算が不正確である場合、アンラップされた2D LUTの最初の「スライス」から常にサンプリングされる可能性がある。これにより、最初のスライスが類似している場合、またはサンプリングがユニークなデータポイントを事実上バイパスしている場合、すべてのLUTが同一に見えるという結果になる。これは、シェーダーがテクスチャを正しく受け取っているにもかかわらず、そのテクスチャをサンプリングするための内部ロジックが壊れており、均一な結果につながることを意味する。

テクスチャユニット	バインドされたWebGLTextureオブジェクトID	関連するLUTファイル名	シェーダーユニフォーム名	ユニフォームロケーション値 (gl.uniform1iから)	Spector.js テクスチャプレビュー
TEXTURE0	(ImageTexture ID)	(ベース画像)	u_image	0	(ベース画像テクスチャ)
TEXTURE1	(LUTTexture ID A)	Anderson.cube	u_lut1	1	(Anderson.cubeテクスチャ)
TEXTURE2	(LUTTexture	Blue	u_lut2	2	(Blue

	ID B)	sierra.cube			sierra.cubeテクスチャ)
TEXTURE3	(LUTTexture ID C)	F-PRO400H.cube	u_lut3	3	(F-PRO400H.cubeテクスチャ)

表：WebGLテクスチャユニットの状態検査
この表は、「シェーダーユニフォームの競合」と「テクスチャの上書き」という中心的な仮説に直接対処する。もしu_lut1とu_lut2の両方がユニフォームロケーション値として0を示す場合、ユニフォームの競合が直ちに確認される。また、「バインドされたWebGLTextureオブジェクトID」と「Spector.jsテクスチャプレビュー」を示すことで、ユニークなテクスチャオブジェクトが実際に異なるユニットにバインドされているか、そして正しいLUTデータがGPU上に存在するかを確認する。

入力ピクセルカラー (RGB)	計算された3D LUT座標 (正規化RGB)	計算された2D UV座標 (アンラップされたテクスチャ上)	シェーダーからのサンプリングされたLUT値	期待される出力カラー (LUT仕様から)
(0.0, 0.0, 0.0)	(0.0, 0.0, 0.0)	(u0, v0)	(r_s0, g_s0, b_s0, a_s0)	(r_e0, g_e0, b_e0, a_e0)
(0.5, 0.5, 0.5)	(0.5, 0.5, 0.5)	(u1, v1)	(r_s1, g_s1, b_s1, a_s1)	(r_e1, g_e1, b_e1, a_e1)
(1.0, 0.0, 0.0)	(1.0, 0.0, 0.0)	(u2, v2)	(r_s2, g_s2, b_s2, a_s2)	(r_e2, g_e2, b_e2, a_e2)

表：フラグメントシェーダー applyLUT サンプル出力
この表は、正しいテクスチャがバインドされていると仮定して、applyLUT関数内の問題を特定するのに役立つ。「計算された3D LUT座標」と「計算された2D UV座標」を期待値と比較することで、3Dから2Dへのマッピングロジックの誤りを迅速に特定できる。もしUV座標が異なる入力に対して一貫して間違っているか同一である場合、それは計算のバグを示している。

C. LUT処理パイプライン (LUTLayer配列、複数LUTの合成)

レイヤー管理システム

LUTLayer配列は、LUTのシーケンスまたはスタックを示唆している。アプリケーションは、この配列を繰り返し処理し、各LUTの変換を適用する必要がある。

もしアプリケーションが複数のLUTを順次適用（連結）する意図がある場合、最初のLUTを介して画像をレンダリング

し、その操作の出力を次のLUTの入力として使用する必要がある。これは通常、オフスクリーンレンダリングのためにフレームバッファオブジェクト（FBO）を使用することを含む [32]。

壊れたマルチパスレンダリングチェーンの可能性: LUTLayerが連結を意味する場合、あるLUT適用の出力が次の入力となる必要がある。これは、FBOにアタッチされたテクスチャにレンダリングし、そのテクスチャを次のシェーダーパスの入力として使用することで行われる [33]。一般的な間違いは、すべての中間結果を正しく渡さずに、すべてのLUTを

同じFBOまたは直接キャンバスにレンダリングすることである。もし各LUTが前のLUTの結果ではなく、元の画像に適用される場合、最後に適用されたLUTのみが可視になるか、効果が意図したとおりに複合されない。もし最後に適用されたLUTが常に同じである（例：他のエラーによるデフォルトまたはアイデンティティLUT）場合、結果は均一になる。これは、異なるLUTがロードされ、バインドされている可能性があっても、その効果が正しく累積されていないため、単一の均一な効果として見えることを意味する。

複数LUTの合成/ブレンド

複数のLUTがブレンドされる（例：不透明度を伴う）場合、シェーダーはブレンドロジックを実装する必要がある [35]。

ブレンドモード（例：乗算、スクリーン、オーバーレイ）と不透明度値は、ユニフォームとしてシェーダーに正しく渡され、LUTごとに適用されなければならない。WebGLのgl.blendFuncとgl.blendFuncSeparateは、ソースカラーとデスティネーションカラーがどのように結合されるかを制御し、しばしばアルファ値を含む [38]。

不正確なブレンドまたは不透明度適用の可能性: 複数のLUTが正しくロードされサンプリングされていても、その合成ロジックに欠陥がある場合、依然として同一に見える可能性がある。例えば、すべてのLUTが不透明度0でブレンドされる場合、またはブレンドモードが事実上「置き換え」操作である場合（例：u_lut1を組み込まずにgl_FragColor = texture2D(u_lut2,...)）、1つのLUTの効果のみが可視になる。もし不透明度値がすべてのLUTに対して常に1.0であり、それらがブレンドなしで順次適用される場合、最後に適用されたもののみが表示される。もし複数のLUTサンプルを結合するためのシェーダーロジックが壊れている場合（例：result = texture2D(lut1) + texture2D(lut2)ではなくresult = blend(texture2D(original), texture2D(lut1)); result = blend(result, texture2D(lut2));）、予期せぬ均一性につながる可能性がある。これは、個々のLUTデータが異なっていても、複数のLUTが関与する場合の最終的なピクセルカラーの計算方法における論理的エラーを示している。

処理シーケンス

パイプラインにおける操作の順序（ロード、テクスチャ作成、バインディング、レンダリング）は、WebGLの状態管理にとって非常に重要である。テクスチャオブジェクトは、テクスチャユニットにバインドされ、描画呼び出しで使用される前に作成され、データが格納されていることを確認する必要がある。非同期画像ロードは、レンダリングがテクスチャのデータが完全にアップロードされる前にテクスチャを使用しようとする競合状態を引き起こす可能性がある [18]。

非同期ロードの競合状態の可能性: LUTファイルは非同期でロードされる。もしレンダリングが、gl.texImage2Dを介

してデータが完全にアップロードされる前にLUTテクスチャを使用しようとする場合、空白または以前の状態にデフォルトで戻る可能性がある。クエリは「LUTが正常にロードされる」と述べているが、これは解析を指す。GPUへのアップロードは依然としてタイミングの問題に影響される可能性がある。もしgl.texImage2Dの呼び出しが最初の描画の後に発生する場合、またはデフォルトの1x1テクスチャがプレースホルダーとして使用されるが [9]、適切に更新されない場合、均一性につながる可能性がある。これは、テクスチャがシェーダーがサンプリングしようとするときに準備ができておらず、結果として一貫してデフォルトまたは不正確なカラーが適用されることを意味する。

D. Canvas2Dフォールバックパイプライン

Canvas2Dプロセッサの分析

Canvas2DはCPU上で動作し、getImageData()を使用してRGBA値のUint8ClampedArrayとして生のピクセルデータを取得し、putImageData()を使用してそれらを書き戻すことで、ピクセルデータを直接操作する [43]。

3D LUTの場合、Canvas2Dプロセッサは次のことを行う必要がある:

1. 入力画像のピクセルデータを取得する。
2. 各ピクセル(R、G、B)に対して、LUT内の対応する3D座標を計算する。
3. 生のLUTデータ(JavaScriptメモリ内にある)に対して三線形補間を実行し、変換されたカラーを取得する。
4. ピクセルデータ配列を新しいカラーで更新する。
5. 変更されたピクセルデータをキャンバスに書き戻す。

Canvas2Dにおけるデータレベルの整合性チェック: もし問題(同一の効果)がCanvas2Dフォールバックでも発生する場合、それは問題がWebGLの状態管理やGPU固有の問題だけではないことを強く示唆している。むしろ、データがWebGLまたはCanvas2Dのいずれかにレンダリングのために渡される前に存在する、LUT適用ロジックにおけるより根本的な欠陥を示している。これは、lutDataMap(または類似のCPU側構造)に格納されているLUTデータ自体が何らかの形で区別されていないか、またはそのデータに対して3Dから2Dへのルックアップと補間を実行するJavaScriptロジックが、すべてのLUTに対して一貫して同じ値をサンプリングしていることを意味する可能性がある。例えば、lutDataMap内のLUTデータ配列がすべて同じ基盤となるArrayBufferを指している場合、またはJavaScriptの補間関数がバグのために常に固定値を返す場合などである。Canvas2Dの動作は重要な診断情報となる。もし同一である場合、問題はGPUとの対話前のlutProcessor.tsロジックにある可能性が高い。

Canvas2Dの制限

Canvas2Dのピクセル操作は本質的にCPUバウンドであり、GPUアクセラレーションされたWebGLよりも著しく遅

い。大規模な画像やリアルタイム効果の場合、性能が低下する [45]。

フォールバックであるとはいえ、Canvas2Dが3D LUTのような複雑なカラー変換に最適化されていないことを認識することが重要である。getImageDataやputImageDataに関連するパフォーマンス警告（例：willReadFrequently）は一般的である [49]。

パフォーマンスと忠実度のトレードオフ: Canvas2DはCPUベースのレンダラーであり、WebGLはGPUを活用する。64^3のLUTを適用する場合、各ピクセルを読み込み、複雑なルックアップ（8点での三線形補間）を実行し、それを書き戻す必要がある。これを大規模な画像に対してCPU上でピクセルごとに実行することは、計算集約的となる。たとえCanvas2DでLUTが最終的に機能したとしても、リアルタイムの画像処理アプリケーションとしてはパフォーマンスが許容できない可能性がある。これは、Canvas2Dフォールバックが主に互換性のためのものであり、コアソリューションは最適なパフォーマンスのためにWebGLパイプラインを優先しつつ、両方のパイプラインで基盤となるデータロジックが健全であることを確認する必要があることを強調している。

V. 根本原因の特定

包括的な分析に基づくと、すべてのLUTが同一のカラー効果を適用する最も可能性の高い根本原因は次のとおりである。

1. シェーダーユニフォームの競合（主要な**WebGL**の原因）: フラグメントシェーダーのsampler2Dユニフォーム（u_lut1、u_lut2、u_lut3）が、JavaScriptのgl.uniform1i呼び出しを介してすべて同じテクスチャユニットインデックス（例：0）に割り当てられている。これにより、シェーダーは、その特定のテクスチャユニットにバインドされている単一のWebGLTextureオブジェクトから一貫してサンプリングすることになり、異なるLUTテクスチャが他のユニットにロードされていても無視される。この状況は、観察された動作の最も直接的な説明となる。ログに異なるLUTデータが正常にロードされていると表示されても、シェーダーが常にTEXTURE0を参照するように指示されている場合、TEXTURE0にバインドされたLUTデータのみが適用される。
2. テクスチャオブジェクトの上書き/共有（副次的な**WebGL**の原因、またはデータ準備の問題）:
    - シナリオ**A**（テクスチャオブジェクトの再利用）: create3DLUTTexture関数（または上位レベルのlutProcessor.tsロジック）が、すべてのLUTデータアップロードに対して、意図せず単一のWebGLTextureオブジェクトを再利用または上書きしている。たとえgl.createTexture()が複数回呼び出されても、lutTextureMap（または類似のキャッシュ）が異なるLUTに対して誤って同じWebGLTextureハンドルを返す場合、または新しいデータでgl.texImage2Dを呼び出す前にgl.bindTextureが常に同じテクスチャオブジェクトに対して呼び出される場合、最後にロードされたLUTのデータのみがGPU上に存在する。
    - シナリオB（データバッファの再利用/破損）: Float32ArrayからUint8Arrayへの変換、またはUint8Arrayバッファ自体が、gl.texImage2Dが呼び出される前に、意図せずすべてのLUTに対して再利用されているか、同一のデータで埋められている。これは、ログで観察された異なるデータが、GPUアップロードのための最終的なデータ準備中に失われていることを意味する。この状況は、Canvas2Dフォールバックでの同一の動作も説明する。
      GPU自体が常に1つのLUTデータセットしか受け取らない場合、異なるLUTが異なる結果を生成することは不可能である。これは、WebGLTextureオブジェクトレベル（ハンドルの再利用）または生データバッファレベル（すべてのテクスチャに対して同じデータをtexImage2Dに渡す）で発生する可能性がある。
      Canvas2Dフォールバックの動作はここで重要であり、もしそれも均一である場合、シナリオBをより強く示唆する。

3. フラグメントシェーダーにおける不正確な**3D→2D**座標マッピング（シェーダーロジックの問題）：applyLUT関数内の、入力RGBカラー（3D座標）をアンラップされたLUTテクスチャ上の2D UV座標にマッピングする内部ロジックに欠陥があり、入力カラーや特定のLUTデータに関係なく、テクスチャの同じ領域から一貫してサンプリングされている。たとえ正しいLUTテクスチャがバインドされ参照されていても、シェーダーがそのテクスチャ内のどこをサンプリングするかという内部ロジックが壊れている場合、常に同じ（または非常に類似した）値を返すため、均一な効果につながる。これはより微妙なバグだが、観察された症状を引き起こす可能性がある。

VI. 解決策の戦略

A. WebGL修正アプローチ

1. 一意の**WebGLTexture**オブジェクトの確保:
   - src/lib/webgl-utils.ts（特にcreate3DLUTTexture）において、新しいLUTが処理されるたびにgl.createTexture()が毎回呼び出され、各LUTに対して真に一意のWebGLTextureオブジェクトが作成されていることを確認する[7]。
   - src/lib/lutProcessor.tsにおいて、lutTextureMapがこれらの一意のWebGLTextureオブジェクトを、LUTファイルから派生した真に一意の識別子（例：コンテンツのハッシュ、またはパスが一意であることが保証されている場合はフルパス）をキーとして正しく格納および取得していることを確認する。
2. 適切なテクスチャユニットの割り当てとユニフォームのバインディング:
   - 各異なるLUT（Anderson.cube、Blue sierra.cube、F-PRO400H.cube）に対して、一意のテクスチャユニットを割り当てる。例えば、ベース画像にはTEXTURE0、Anderson.cubeにはTEXTURE1、Blue sierra.cubeにはTEXTURE2、F-PRO400H.cubeにはTEXTURE3を使用する。
   - 描画前に、正しいテクスチャユニットをアクティブにし、対応する一意のWebGLTextureオブジェクトをバインドする。
     TypeScript
     ```
     gl.activeTexture(gl.TEXTURE0 + 1); // u_lut1用
     gl.bindTexture(gl.TEXTURE_2D, lutTextureMap.get('Anderson.cube'));
     gl.activeTexture(gl.TEXTURE0 + 2); // u_lut2用
     gl.bindTexture(gl.TEXTURE_2D, lutTextureMap.get('Blue sierra.cube'));
     //... TEXTURE3についても同様
     ```

     [7]
   - フラグメントシェーダーのユニフォーム（u_lut1、u_lut2、u_lut3）を、それぞれ対応するテクスチャユニットインデックスを指すように更新する。
     TypeScript
     ```
     const u_lut1Location = gl.getUniformLocation(program, "u_lut1");
     gl.uniform1i(u_lut1Location, 1); // TEXTURE1を指す
     const u_lut2Location = gl.getUniformLocation(program, "u_lut2");
     ```

```
gl.uniform1i(u_lut2Location, 2); // TEXTURE2を指す
//... u_lut3Location, 3についても同様
```

18

- もしサンプラーの配列（例：uniform sampler2D u_luts[N];）を使用している場合、WebGL2コンテキストがアクティブであり、動的インデックス付けがサポートされていることを確認するか、上記のように個別のユニフォームにリファクタリングする [17]。

3. **applyLUTと複数LUT合成ロジックの改善:**
   - **3D→2D座標マッピング:** フラグメントシェーダーのapplyLUT関数内のGLSLコードを注意深く見直し、3D RGB座標からアンラップされたテクスチャ上の2D UV座標への正確なマッピングを確認する。正確なサンプリングのために、正規化とスライス計算、および潜在的なハーフテクセルオフセットを確実に含める [8]。
   - **三線形補間:** 手動で三線形補間が実装されている場合（3Dデータを表現する2Dテクスチャの場合）、その正確性を検証する。これには8つの点をサンプリングし、線形補間を行うことが含まれる [3]。
   - **複数LUTの連結/ブレンド:** 複数のLUTが順次適用される場合、フレームバッファオブジェクト（FBO）を使用したマルチパスレンダリング戦略を実装する。最初のLUTで元の画像をFBOにレンダリングし、そのFBOのテクスチャを次のLUTの入力として使用し、最終結果がキャンバスにレンダリングされるまでこれを繰り返す [32]。ブレンドが望ましい場合、各LUTレイヤーに対してシェーダー内でブレンドモードと不透明度値が正しく適用されていることを、gl.blendFuncまたはgl.blendFuncSeparateと適切なユニフォーム値を使用して確認する [35]。

B. Canvas2D互換の修正アプローチ

1. **CPU側LUTデータの整合性検証:** ピクセル操作を行う前に、lutDataMap（または解析されたLUTデータを保持する類似の構造）に、各LUTに対して真に異なるUint8ArrayまたはFloat32Arrayバッファが含まれていることを確認する。もしCanvas2Dフォールバックでも同一の効果が見られる場合、これはデータ自体がこの段階で均一になっていることを強く示唆している。
2. ピクセルごとの**LUT**適用ロジックの修正: Canvas2Dプロセッサにおいて、ピクセルを繰り返し処理し、3D LUTのルックアップを実行し、変換を適用するJavaScriptロジックが、異なるLUTデータ配列を正しく参照していることを確認する。ルックアップと補間のロジックは、概念的にGLSLのapplyLUT関数を反映しているべきである。
   - ctx.getImageData()を使用して生のピクセル配列を取得する。
   - 各(R, G, B)ピクセルに対して、3D LUT座標を計算する。
   - CPU側のLUTデータに対して三線形補間を実行する。
   - ピクセル配列を更新する。
   - ctx.putImageData()を使用して変更をキャンバスに適用する [43]。
3. パフォーマンスに関する考慮事項: Canvas2Dは、大規模な画像に対する64^3のLUTの場合、著しく遅くなることを認識する。もしgetImageDataが繰り返し呼び出される場合、getContext('2d')にwillReadFrequently: trueヒントを追加することを検討するが、これは小さな最適化に過ぎない [49]。

VII. デバッグ手法

A. ブラウザ開発者ツール (Chrome DevTools)

- **WebGL**パネル: Chrome DevToolsの「WebGL」パネル（「その他のツール」→「レンダリング」または「グラフィックス」の下）を有効にする。このパネルでは、アクティブなテクスチャ、バインドされたバッファ、ユニフォーム値など、現在のWebGLの状態を検査できる [50]。
- シェーダー検査: コンパイルされたフラグメントシェーダーを検査する。コンソールに報告されない予期せぬ最適化やエラーがないかを確認する [50]。
- パフォーマンスタブ: フレームレートとGPU使用率を監視し、ボトルネックを特定する [50]。
- メモリタブ: テクスチャが期待どおりに割り当てられ、解放されていることを確認するためにGPUメモリ使用量をチェックする。

B. Spector.js

- フレームキャプチャと検査: Spector.js Chrome拡張機能をインストールするか（フレームワークに組み込まれている場合はそのバージョンを使用）、[53]、フレームをキャプチャしてすべてのWebGL呼び出しの詳細なトレースを取得する [50]。
- テクスチャコンテンツビューア: Spector.jsは、GPU上のテクスチャの実際のピクセルデータを表示する上で非常に貴重である。これにより、異なるLUTデータが実際に一意のWebGLTextureオブジェクトにアップロードされていることを確認できる。描画コマンドを選択し、使用されているテクスチャを検査する [55]。
- ユニフォーム検査: 各描画呼び出しにおけるユニフォーム変数（u_lut1、u_lut2など）の値を検査し、それらが異なるテクスチャユニットを正しく指していることを確認する。これにより、「シェーダーユニフォームの競合」を直接確認できる [50]。

C. コードレベルのデバッグ

- 戦略的な**console.log:** src/lib/lutProcessor.tsとsrc/lib/webgl-utils.tsの主要なポイントにconsole.logステートメントを挿入し、以下の情報をログに出力する。
  - 各LUTファイルについて、解析後の生解析LUTデータ（例：Float32Arrayの最初の数値を）をログに記録する。
  - 各LUTについて、gl.createTexture()の戻り値であるWebGLTextureオブジェクト参照をログに記録し、一意性を確認する。

    - 各LUTについて、gl.texImage2D呼び出しの直前のUint8Arrayデータをログに記録する。
    - gl.activeTexture、gl.bindTexture、gl.uniform1i呼び出しの直前のテクスチャユニットインデックスとユニフォームロケーションをログに記録する。
- **JavaScriptデバッガー:** ブラウザのJavaScriptデバッガーでブレークポイントを使用し、lutProcessor.tsとwebgl-utils.tsのコードをステップ実行し、テクスチャ作成、データ変換、バインディングの各段階で変数の値を検査する。

D. シェーダーにおける視覚デバッグ

フラグメントシェーダーを一時的に変更し、診断用のカラーを出力させる。例えば:

- u_lut1がアクティブであることを確認するため:gl_FragColor = texture2D(u_lut1, v_texcoord);(LUT1の効果が表示されるはず)。
- u_lut2を確認するため:gl_FragColor = texture2D(u_lut2, v_texcoord);(LUT2の効果が表示されるはず)。
- LUTサンプリングの入力座標を可視化するため:gl_FragColor = vec4(inputColor.rgb, 1.0);(元のカラーが表示される)。
- アンラップされたLUTの計算された2D UV座標を可視化するため:gl_FragColor = vec4(uv_coords.x, uv_coords.y, 0.0, 1.0);(グラデーションが表示され、マッピングエラーが明らかになる)。

VIII. 実装ロードマップ

1. フェーズ1:診断とWebGLコア修正(高優先度)

- ステップ**1.1**:一意の**WebGLTexture**作成の検証**:**
    - src/lib/webgl-utils.ts::create3DLUTTextureを見直し、新しいLUTごとにgl.createTexture()が常に呼び出されていることを確認する。
    - gl.createTexture()によって返されるWebGLTextureオブジェクトをログに出力し、lutTextureMapに保存するconsole.logステートメントを追加する。各LUTが異なるオブジェクトを取得していることを検証する。
- ステップ**1.2**:テクスチャユニットバインディングとユニフォーム割り当ての修正**:**
    - src/lib/lutProcessor.ts(またはレンダリングループ)を修正し、各一意のWebGLTextureオブジェクトが異なるテクスチャユニット(例:gl.TEXTURE0 + 1、gl.TEXTURE0 + 2など)にバインドされていることを確認する。
    - フラグメントシェーダー内の対応するsampler2Dユニフォーム(u_lut1、u_lut2、u_lut3)が、gl.uniform1iを

使用してこれらの異なるテクスチャユニットインデックスに正しく設定されていることを確認する。
    - もしサンプラーの配列を使用している場合、WebGL2コンテキストがアクティブであり、動的インデックス付けがサポートされていることを検証するか、個別のユニフォームにリファクタリングする。
- ステップ**1.3**：**3D→2D**座標マッピングの検証（フラグメントシェーダー）：
    - フラグメントシェーダーのapplyLUT関数を注意深く見直す。
    - gl_FragColorにvec4(uv.x, uv.y, 0.0, 1.0)またはvec4(inputColor.rgb, 1.0)を出力することで視覚デバッグを実装し、正しい座標生成と入力カラーの渡しを確認する。
- ステップ**1.4**：テストと検証：各修正後、アプリケーションを再実行し、Spector.jsを使用してテクスチャの内容、ユニフォーム値、および全体的なレンダリングパスを検査し、異なるLUT効果が確認できることを検証する。

2. フェーズ2：複数LUT合成とCanvas2Dフォールバック（中優先度）

- ステップ**2.1**：マルチパスレンダリングの実装（連結の場合）：
    - 複数のLUTを連結する意図がある場合、FBOを使用して中間パスのレンダリングパイプラインをリファクタリングする。
    - 各LUT適用は新しいテクスチャにレンダリングされ、それが次のパスの入力となるようにする。
- ステップ**2.2**：ブレンドロジックの検証（ブレンドの場合）：
    - シェーダーのブレンドロジックと、不透明度およびブレンドモードのJavaScriptユニフォーム設定を見直す。
    - gl.blendFuncまたはgl.blendFuncSeparateが目的の視覚効果のために正しく設定されていることを確認する。
- ステップ**2.3**：**Canvas2D**フォールバックのデバッグ：
    - もしCanvas2Dで問題が続く場合、src/lib/lutProcessor.tsのCPU側LUT適用ロジックに焦点を当てる。
    - ピクセル操作前のUint8Arrayデータと、サンプルピクセルに対するLUTルックアップの結果を検査するためにconsole.logステートメントを追加する。
    - 3D→2D座標マッピングと三線形補間のJavaScript実装の正確性を検証する。

3. フェーズ3：最適化と改善（低優先度、修正後）

- アンラップされたLUTが非2のべき乗（NPOT）テクスチャである場合に備え、適切なテクスチャサンプリングを確実にするために、テクスチャパラメータ設定（gl.texParameteri）をCLAMP_TO_EDGEとLINEARフィルタリングに実装する[9]。
- パフォーマンスが重要でWebGL2が保証されている場合、ネイティブ3D LUTサポートのためにWebGL2のTEXTURE_3Dの使用を検討する[6]。

IX. 結論

本調査は、WebGLのテクスチャユニット管理とユニフォーム変数割り当てにおける重大な設定ミスが、LUTの同一効果の主要な原因であることを示している。これらの領域を体系的に対処し、シェーダーロジックと複数LUTの合成を慎重に検証することで、アプリケーションは異なるカラー変換を正しくレンダリングできるようになる。特にSpector.jsを活用したGPU状態の検査を含む、概説されたデバッグ手法は、修正の検証に不可欠となる。問題が解決されれば、アプリケーションは意図された多様な視覚効果を提供し、画像処理能力が向上する。将来の考慮事項としては、長期的な安定性と応答性を確保するための堅牢なエラー処理とパフォーマンスプロファイリングを含めるべきである。

引用文献

1. The Essential Guide to LUTs - Frame.io Insider, 6月 19, 2025にアクセス、https://blog.frame.io/2019/08/12/luts-101/
2. Color Correction lookup texture - Unity - Manual, 6月 19, 2025にアクセス、https://docs.unity3d.com/550/Documentation/Manual/script-ColorCorrectionLookup.html
3. Three-Dimensional Lookup Table with Interpolation - SPIE, 6月 19, 2025にアクセス、https://spie.org/samples/PM159.pdf
4. Limitations of Custom LUTs : r/colorists - Reddit, 6月 19, 2025にアクセス、https://www.reddit.com/r/colorists/comments/119tbvu/limitations_of_custom_luts/
5. 3D texture in WebGL/three.js using 2D texture workaround? - Stack Overflow, 6月 19, 2025にアクセス、https://stackoverflow.com/questions/23006414/3d-texture-in-webgl-three-js-using-2d-texture-workaround
6. ColorGradingTexture - Babylon.js Documentation, 6月 19, 2025にアクセス、https://doc.babylonjs.com/typedoc/classes/BABYLON.ColorGradingTexture
7. WebGL2 Using 2 or More Textures, 6月 19, 2025にアクセス、https://webgl2fundamentals.org/webgl/lessons/webgl-2-textures.html
8. My take on shaders: Color grading with Look-up Textures (LUT ..., 6月 19, 2025にアクセス、https://halisavakis.com/my-take-on-shaders-color-grading-with-look-up-textures-lut/
9. WebGL Textures, 6月 19, 2025にアクセス、https://webglfundamentals.org/webgl/lessons/webgl-3d-textures.html
10. WebGLRenderingContext: texImage2D() method - Web APIs | MDN, 6月 19, 2025にアクセス、https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/texImage2D
11. WebGL Shaders and GLSL, 6月 19, 2025にアクセス、https://webglfundamentals.org/webgl/lessons/webgl-shaders-and-glsl.html
12. How video games use LUTs and how you can too - FrostKiwi's Secrets, 6月 19, 2025にアクセス、https://blog.frost.kiwi/WebGL-LUTS-made-simple/
13. WebGL2 3D Perspective Correct Texture Mapping, 6月 19, 2025にアクセス、https://webgl2fundamentals.org/webgl/lessons/webgl-3d-perspective-correct-te

xturemapping.html
14. Using textures in WebGL - Web APIs | MDN, 6月 19, 2025にアクセス、https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial/Using_textures_in_WebGL
15. Unwrapping a polygon from 3d to 2d space (using triangles) for texture coordinates, 6月 19, 2025にアクセス、https://stackoverflow.com/questions/4845175/unwrapping-a-polygon-from-3d-to-2d-space-using-triangles-for-texture-coordinat
16. Trilinear interpolation in 3d textures - OpenGL: Advanced Coding - Khronos Forums, 6月 19, 2025にアクセス、https://community.khronos.org/t/trilinear-interpolation-in-3d-textures/58740
17. Introduction to Computer Graphics, Section 6.4 -- Image Textures, 6月 19, 2025にアクセス、https://math.hws.edu/eck/cs424/graphicsbook-1.2/c6/s4.html
18. WebGL Using 2 or More Textures, 6月 19, 2025にアクセス、https://webglfundamentals.org/webgl/lessons/webgl-2-textures.html
19. How to send multiple textures to a fragment shader in WebGL? - Stack Overflow, 6月 19, 2025にアクセス、https://stackoverflow.com/questions/19755320/how-to-send-multiple-textures-to-a-fragment-shader-in-webgl
20. WebGLRenderingContext: getUniformLocation() method - Web APIs | MDN, 6月 19, 2025にアクセス、https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/getUniformLocation
21. WebGL - Drawing Multiple Things, 6月 19, 2025にアクセス、https://webglfundamentals.org/webgl/lessons/webgl-drawing-multiple-things.html
22. Texture Units - WebGL Fundamentals, 6月 19, 2025にアクセス、https://webglfundamentals.org/webgl/lessons/webgl-texture-units.html
23. WebGL2 Image Processing, 6月 19, 2025にアクセス、https://webgl2fundamentals.org/webgl/lessons/webgl-image-processing.html
24. WebGLRenderingContext: bindTexture() method - Web APIs | MDN, 6月 19, 2025にアクセス、https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/bindTexture
25. WebGL best practices - Web APIs - MDN Web Docs - Mozilla, 6月 19, 2025にアクセス、https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/WebGL_best_practices
26. WebGL Text - Textures, 6月 19, 2025にアクセス、https://webglfundamentals.org/webgl/lessons/webgl-text-texture.html
27. WebGL and OpenGL - WebGL Public Wiki, 6月 19, 2025にアクセス、https://www.khronos.org/webgl/wiki/WebGL_and_OpenGL
28. WebGL Texture read/write at the same time - Stack Overflow, 6月 19, 2025にアクセス、https://stackoverflow.com/questions/9271149/webgl-texture-read-write-at-the-s

ame-time
29. WebGL/sdk/tests/conformance/glsl/bugs/sampler-array-using-loop-index.html at main, 6月 19, 2025にアクセス、https://github.com/KhronosGroup/WebGL/blob/master/sdk/tests/conformance/glsl/bugs/sampler-array-using-loop-index.html
30. GLSL, Array of textures of differing size - Stack Overflow, 6月 19, 2025にアクセス、https://stackoverflow.com/questions/12030711/glsl-array-of-textures-of-differing-size
31. WebGL: How to bind an array of samplers - javascript - Stack Overflow, 6月 19, 2025にアクセス、https://stackoverflow.com/questions/17093477/webgl-how-to-bind-an-array-of-samplers
32. How To COMBINE Multiple LUTs Into ONE (So Helpful!) - YouTube, 6月 19, 2025にアクセス、https://www.youtube.com/watch?v=K0JSd8R4ao8
33. javascript - WebGL - apply several programs successively - Stack Overflow, 6月 19, 2025にアクセス、https://stackoverflow.com/questions/36991367/webgl-apply-several-programs-successively
34. canvas - WebGL - two pass rendering - Stack Overflow, 6月 19, 2025にアクセス、https://stackoverflow.com/questions/42162228/webgl-two-pass-rendering
35. Blending (using WebGL/Three.js) - And How to Combine Additive and Alpha Blending, 6月 19, 2025にアクセス、https://www.youtube.com/watch?v=AxopC4yW4uY
36. How to merge two or more LUTs into one LUT in 3D LUT Creator - YouTube, 6月 19, 2025にアクセス、https://www.youtube.com/watch?v=m1vX_CyI7FY
37. Blend Modes - Maxim McNair, 6月 19, 2025にアクセス、https://maximmcnair.com/p/webgl-blend-modes
38. WebGL, Blending, and Why You're Probably Doing it Wrong - Limnu, 6月 19, 2025にアクセス、https://limnu.com/webgl-blending-youre-probably-wrong/
39. Alpha Blending in WebGL - Blog, 6月 19, 2025にアクセス、http://www.delphic.me.uk/tutorials/webgl-alpha
40. Blending - LearnOpenGL, 6月 19, 2025にアクセス、https://learnopengl.com/Advanced-OpenGL/Blending
41. WebGL: Texture rendering not working correctly - Stack Overflow, 6月 19, 2025にアクセス、https://stackoverflow.com/questions/43005778/webgl-texture-rendering-not-working-correctly
42. Textures and Web GL errors. - HTML5 Game Devs Forum, 6月 19, 2025にアクセス、https://www.html5gamedevs.com/topic/19381-textures-and-web-gl-errors/
43. Pixel manipulation with canvas - Web APIs | MDN, 6月 19, 2025にアクセス、https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Pixel_manipulation_with_canvas
44. HTML5 Canvas/Pixel manipulation & Animations, 6月 19, 2025にアクセス、https://pictureelement.github.io/html5tech/canvas-pixel-manipulation-and-animations.html

45. CanvasRenderingContext2D: putImageData() method - Web APIs | MDN, 6月 19, 2025にアクセス、https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/putImageData
46. Image manipulation techniques with 2d canvas - MadeByMike, 6月 19, 2025にアクセス、https://www.madebymike.com.au/writing/canvas-image-manipulation/
47. Optimizing canvas - Web APIs | MDN, 6月 19, 2025にアクセス、https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Optimizing_canvas
48. Improving HTML5 Canvas performance | Articles - web.dev, 6月 19, 2025にアクセス、https://web.dev/articles/canvas-performance
49. How to Boost Image Performance When Changing Brightness - Bokeh Discourse, 6月 19, 2025にアクセス、https://discourse.bokeh.org/t/how-to-boost-image-performance-when-changing-brightness/12415
50. Best Practices for Testing and Debugging WebGL Applications - PixelFreeStudio Blog, 6月 19, 2025にアクセス、https://blog.pixelfreestudio.com/best-practices-for-testing-and-debugging-webgl-applications/
51. How to Optimize WebGL for High-Performance 3D Graphics, 6月 19, 2025にアクセス、https://blog.pixelfreestudio.com/how-to-optimize-webgl-for-high-performance-3d-graphics/
52. From WebGL to WebGPU | Blog - Chrome for Developers, 6月 19, 2025にアクセス、https://developer.chrome.com/blog/from-webgl-to-webgpu
53. WebGLRenderer - What is Phaser?, 6月 19, 2025にアクセス、https://docs.phaser.io/api-documentation/class/renderer-webgl-webglrenderer
54. Spector.js - Chrome Web Store, 6月 19, 2025にアクセス、https://chromewebstore.google.com/detail/spectorjs/denbgaamihkadbghdceggmchnflmhpmk
55. How can I inspect or dump texture contents in WebGL? - Stack Overflow, 6月 19, 2025にアクセス、https://stackoverflow.com/questions/34769866/how-can-i-inspect-or-dump-texture-contents-in-webgl
56. What is the best way to debug a webgl program? - Reddit, 6月 19, 2025にアクセス、https://www.reddit.com/r/webgl/comments/19bemj3/what_is_the_best_way_to_debug_a_webgl_program/