These Colours Do (Not) Exist: Rendering 3-Dimensional Colour Spaces Using Modern Graphics Pipelines

Kyle Lukaszek

1113798

BComp (Hons) Computer Science

School of Computer Science

University of Guelph

2024

Abstract

This paper explores the intersection of modern graphics technology and colour science, presenting a modern approach to colour space visualization using WebGPU and compute shader optimization. While existing solutions often require expensive specialized software or lack real-time interaction capabilities, our implementation leverages point-based rendering techniques and parallel computation to achieve real-time performance on consumer hardware. The solution demonstrates how modern graphics APIs can be utilized to create responsive, interactive visualizations of complex colour spaces through web browsers. Real-time interaction capabilities enable new possibilities for both educational applications and scientific analysis, allowing users to explore colour relationships and transformations dynamically. We begin with a comprehensive examination of colorimetry fundamentals, establishing the theoretical framework that underpins colour perception and measurement. Building on this foundation, we detail our WebGPU-based visualization approach, which achieves significant rendering speed improvements compared to traditional CPU-based methods. Performance benchmarks demonstrate the effectiveness of our implementation in delivering professional-grade accuracy while maintaining accessibility. This work contributes to democratizing colour space visualization tools for both educational and professional applications.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction to Colorimetry

Visualizing **3-dimensional colour spaces** presents a unique challenge in modern computer graphics. Despite significant advances in rendering technology, the scientific community continues to rely on tooling that falls short of modern rendering standards. Research solutions such as MATLAB's colour space visualization tool set, though mathematically robust, suffer from limited rendering sophistication due to their reliance on OpenGL 4.0/WebGL [20]. This gap between available tools and modern rendering capabilities has implications for research efficacy and knowledge dissemination in colorimetry. However, it is important to develop a deep understanding of what we are trying to render before we address these technical limitations and propose a more sophisticated solution. While there might be infinite possible combinations of wavelengths in the visible spectrum, human perception of these wavelengths is constrained by biological limitations and environmental factors [18]. These phenomenons underscore why we need precise mathematical frameworks—**colour gamuts** and **colour spaces**—to quantify colour relationships objectively. This chapter establishes the theoretical foundation of **colorimetry** necessary for understanding the rendering approaches discussed in subsequent chapters. By examining the physical, physiological, and perceptual aspects of colour, we will better appreciate the complexities involved in visualizing **3-dimensional colour spaces** using modern rendering techniques.

1.1 What Is Colour?

Colour can be defined as a visual phenomenon that emerges from our governing mechanics. That is to say, colour exists due to the intersection of various scientific fields of study ranging from physics to biology to psychology.

1.1.1 Physical Aspect of Colour

The first step in understanding colour is understanding its physical basis.

1. Visible light is electromagnetic radiation with wavelengths between 380-780nm. (Fig 1.1)[16]

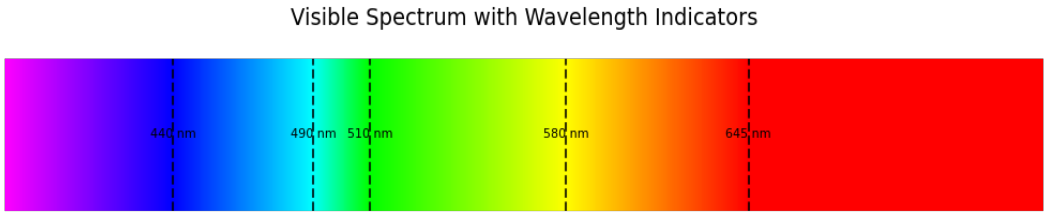

Figure 1.1: Colors According To Wavelengths

2. Non-visible light exists outside of this range and can be seen by some animals. These are called ultraviolet and infrared rays.

3. These wavelengths are transformed as they bounce around the environment. This causes objects to appear coloured by selectively absorbing, reflecting, or emitting different wavelengths of light.

4. Wavelengths can be controlled and emitted using chemical and electrical engineering. For example, most displays use minuscule liquid crystal semiconductors to draw pixels to the screen through some sophisticated chemistry and physics. [23]

1.1.2 Physiological Aspect of Colour

The next step in understanding colour is grasping the physiological mechanism of colour. Not only are we perceiving colour, but our eyes are performing many tasks subconsciously to ensure that all light reflected into them gets translated into an image in our minds. The specifics aren't important for our purposes, but the following points should be enough information to familiarize yourself with how humans perceive colour.

1. Our eyes contain retinal photoreceptors, also known as **rods** and **cones**.
2. Rods are very sensitive to light and are used for processing vision in low-light conditions.
3. Cones are less sensitive to light but instead, they are highly sensitive to the electromagnetic radiation that makes up "colour".
4. Each cone is sensitive to a specific colour. The S-Cones (β) are sensitive to blue wavelengths, M-Cones (γ) are sensitive to green wavelengths, and L-cones (ρ) are sensitive to red wavelengths. (Fig 1.2)

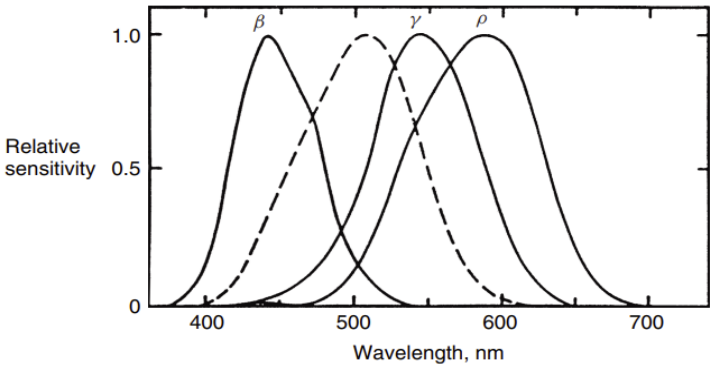

Figure 1.2: Relative Sensitivities of Retinal Photoreceptors [16]

5. Cones then pass off the incoming light information to photoreceptor channels so that our mind can conjure what is in front of us. Signals are processed

through **red-green channels, blue-yellow channels, and an achromatic channel (light-dark)**. [16]

1.1.3 Perceptual Attributes of Colour

Although everyone sees colour differently, there so happens a set of attributes that a large percentage of the population can agree upon when it comes to colour identification. These are commonly referred to as the perceptual attributes of colour and are defined as follows [16]:

1. ***Brightness*:** How bright or dim a given area appears.
2. ***Hue*:** A wavelength-dependent quality that measures the similarity to one, or a combination of two, of red, green, blue, and yellow, in a given area.
3. ***Colourfulness*:** Quantifies the degree to which a given area will exhibit more or less hue.
4. ***Saturation*:** Relation between the colourfulness and brightness of the given area. If a colour is said to be saturated, then the wavelengths must have high spectral purity.
5. ***Chromaticity*:** The measure of colour intensity of a given area, independent of brightness and hue, relative to another similarly illuminated area. See 1.3 for a more visual explanation of chromaticity. The calculations for this are rather long, but they can be found in the International Commission on Illumination's (CIE) report on *Colorimetry* [5].
6. ***Lightness*:** The measure of brightness of a given area, relative to the brightness of another similarly illuminated area (e.g. light red V.S. dark red). The brightness will be directly correlated to the illuminant.

The perceptual attributes of colour are important as they are the primary metrics used to quantify colour spaces outside simple colour spaces such as RGB. These attributes can be objectively measured using instruments such as spectroradiomet-

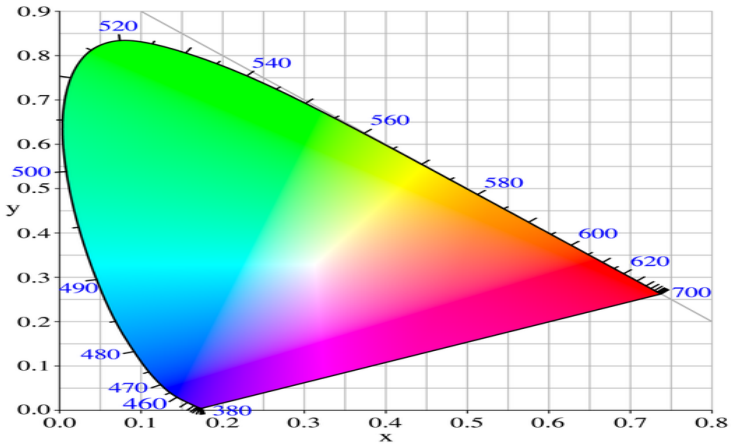

Figure 1.3: CIE1931 xy Chromaticity Chart
[9]

ers. According to Hunt and Pointer [16], the mean colour difference between a given sample, and the meticulously crafted and revised test set used for instrument verification, normally falls within a radius of 0.1 units (or less) of a colour difference unit in a **perceptually uniform colour space**. Given that we are capable of accurately measuring the wavelengths travelling throughout space, how do we represent colour in a more *intuitive* manner? This leads us to our next topic that needs to be discussed: **colour spaces**.

Chapter 2

Modelling Colour Spaces

Given that there are attributes we can use to quantify colour, that would mean that there must be some abstract way to think of colour outside of what our eyes are telling us. Now let us introduce the concept of **colour spaces**. A colour space is defined as a set of all possible colours that can be represented using a specific set of parameters and serves as a mathematical framework for quantifying colour [16]. A key attribute of a colour space is its linearity. The linearity of colour space depends on whether or not it follows standard colour-blending logic.

2.1 Linear Colour Spaces

Some sources refer to linear colour spaces as additive colour spaces.

The most common linear colour space is the RGB colour space, which represents colours using three parameters corresponding to red, green, and blue light intensities. This space is fundamental to light-based rendering systems, including displays, LEDs, and computer graphics. We can formally define the RGB colour space as:

$$RGB_{linear} = \{(R,\ G,\ B) \in \mathbb{R}^3 \mid R,\ G,\ B \in [0.0,\ 1.0]\} \tag{2.1}$$

In this representation, each component (R, G, B) represents the raw intensity of its respective colour channel. This linear relationship between numerical values and light intensity makes the space suitable for physical light calculations. For instance, $RGB_{linear} = (0.0,\ 0.0,\ 0.0)$ represents the absence of light (black), while $RGB_{linear} = (1.0,\ 0.0,\ 0.0)$

represents maximum red intensity with no green or blue contribution. This means that white is represented when $RGB_{linear} = (1.0,\ 1.0,\ 1.0)$.

While theoretically defined over real numbers in $[0.0,\ 1.0]$, practical implementations typically quantize these values to integers in $[0,\ 255]$ (See fig 2.1), giving us:

$$RGB_{linear} = \{(R,\ G,\ B) \in \mathbb{Z}^3 \mid R,\ G,\ B \in [0, 255]\} \tag{2.2}$$

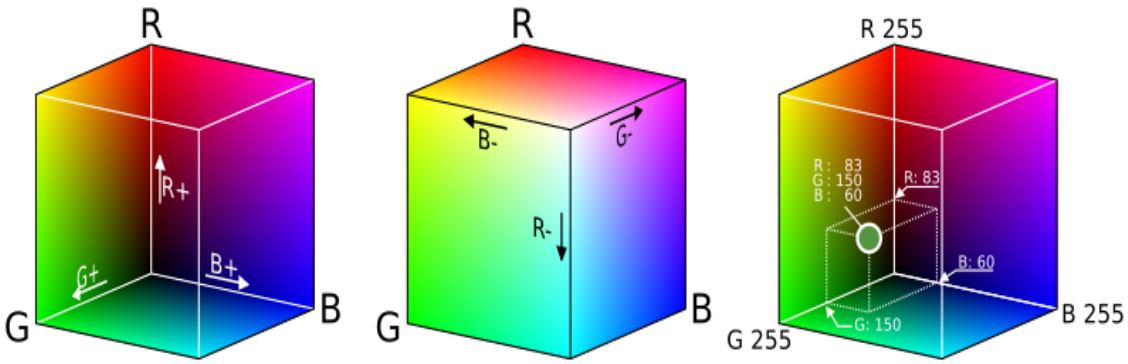

Figure 2.1: Visual Representation of the RGB Colour Space [10]

This quantization from $[0.0,\ 1.0] \in \mathbb{R}$ to $[0,\ 255] \in \mathbb{Z}$ reflects a crucial practical consideration in computer graphics: memory efficiency. By encoding each channel as an 8-bit unsigned integer instead of a 32-bit floating-point number, we reduce the memory footprint of each pixel by 75%. This 24-bit RGB format has become the de facto standard for most applications, with higher bit depths (like 30-bit) reserved for HDR-compatible hardware. The discrete nature of this representation naturally limits the number of representable colours.

For the standard 24-bit format:

$$\text{24-Bit RGB Colour: } P(2^8,\ 3) = 2^8 \times 2^8 \times 2^8 = 16581375 \textbf{ representable colours} \tag{2.3}$$

While HDR10's 30-bit format provides:

$$\text{30-bit RGB Colour: } P(2^{10},\ 3) = 2^{10} \times 2^{10} \times 2^{10} = 1073741824 \textbf{ representable colours} \tag{2.4}$$

Though convenient for computation, this colour space presents a significant limitation: it fails to account for human perception of light intensity. While our hardware can represent millions of distinct colours, our eyes perceive differences in brightness non-linearly. This perceptual gap motivates the development of non-linear colour spaces, which we will explore in the next section.

2.2 Non-Linear Colour Spaces

Non-linear colour spaces diverge from the direct representational model we explored in linear colour spaces. Rather than representing colours as simple combinations of primary colours, these spaces employ mathematical transformations to map colours in ways that better align with human perception [21] or address technical limitations. The most widely adopted of these spaces is **sRGB**, which has become the de facto standard for web content and digital displays [2].

sRGB

Although both sRGB and linear RGB make use of 8-bit colour channels, they represent fundamentally different approaches to colour encoding. Where linear RGB maintains strict proportionality between numerical values and light intensity, sRGB proposes a standardized non-linear transformation that better matches human perception of brightness differences [17].

To convert from linear RGB to sRGB, we first normalize our linear RGB values to the range $[0.0,\ 1.0] \in \mathbb{R}$:

$$RGB_{norm} = \frac{RGB_{linear}}{2^8},\ \text{where } 2^8 = 255 \text{ is the bit depth per channel} \tag{2.5}$$

The conversion to sRGB applies a piecewise function, performing gamma correction on each normalized colour component (C_{linear}):

$$C_{\text{sRGB}} = \begin{cases} C_{\text{linear}} \cdot 12.92, & \text{if } C_{\text{linear}} < 0.0031308, \\ 1.055 \cdot C_{\text{linear}}^{0.41666} - 0.055, & \text{if } C_{\text{linear}} \geq 0.0031308. \end{cases} \quad [17] \tag{2.6}$$

Therefore,

$$sRGB_{norm} = \{(C_R,\ C_G,\ C_B) \in \mathbb{R}^3 \mid C_R,\ C_G,\ C_B \in [0.0, 1.0]\}$$

We can then undo the normalization process by performing $round(sRGB_{norm} \cdot 255)$ to find that:

$$sRGB = \{(R,\ G,\ B) \in \mathbb{Z}^3 \mid R,\ G,\ B \in [0, 255]\}$$

This definition is identical to the definition of RGB! What gives? If we think about the math for just a second, we realize that the first case of the piecewise function covers nearly 100% of all colours we can represent. What does this mean? Well, let us think back to the CIE1931 chromaticity diagram (1.3) for just a second (which is also a non-linear colour space!).

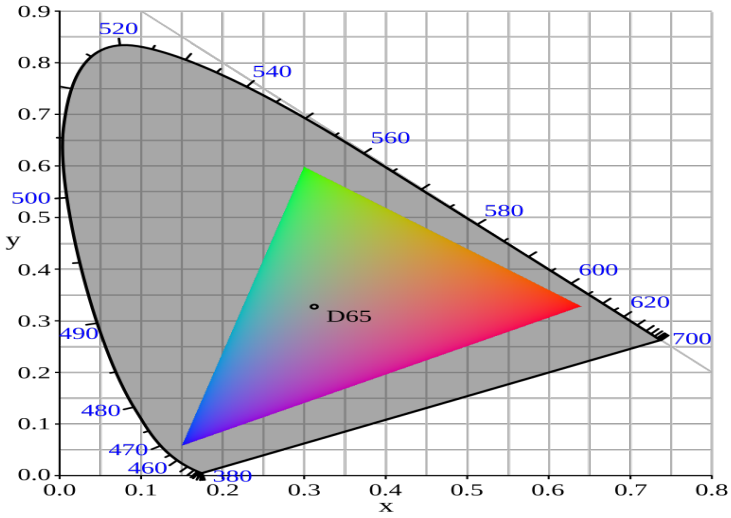

Figure 2.2: sRGB colour space imposed on the CIE1931 chromaticity diagram [11]

Optimizing Non-Linear Colour Representation

The CIE's research on spectral weighting functions enabled the mapping of visible colours under specific illuminants (white points) through values x and y [5]. This mapping, known as CIE1931, reveals an important insight to us: while standard 24-bit displays can represent approximately 16 million colours, many of these numerical combinations correspond to

colour differences that are barely perceptible to the human eye [30]. The sRGB standard addresses this inefficiency through strategic compression of the colour space.

As illustrated in figure 2.3, the sRGB colour space forms a triangular subset within the CIE1931 chromaticity diagram. This deliberate limitation of the colour space, known as a **colour gamut**, enables more efficient distribution of the available colour values across perceptually meaningful differences [15]. The widespread adoption of sRGB among display manufacturers and content creators has established it as the standard colour gamut for digital content reproduction. Nevertheless, broader colour spaces remain crucial for professional applications. While sRGB serves well for final content delivery, maintaining colour precision during capture, creation, and processing often requires more expansive colour representations. This is particularly relevant in computer graphics and image processing, where the computational cost of working with larger colour spaces is negligible on modern hardware [15]. Modern display systems handle the conversion between different bit depths (such as 24-bit to 30-bit) at the driver level, freeing creatives to work in whatever colour space best suits their needs. This enhanced flexibility in colour representation sets the stage for our final topic of the chapter...

2.3 Perceptually Uniform Colour Spaces

Colour spaces such as HSL, HSV, and CIELUV utilize the perceptual attributes of colour to define a colour space (See 1.1.3). This allows for a more intuitive understanding of colour spaces, as they are directly related to how humans perceive colour. These spaces are known as **perceptually uniform colour spaces** and are purposefully designed to better represent colour in a way more representative of how humans physically perceive colour.

2.3.1 CIELUV

The CIELUV space, in particular, represents a significant advancement in perceptually uniform colour representation. Unlike RGB-based spaces, CIELUV encodes colour using three parameters that directly correspond to human perception: $L*$ for lightness, $u*$ for the

red-green axis, and $v*$ for the blue-yellow axis [5]. The conversion from CIEXYZ (results of spectral weighting functions) to CIELUV begins with the calculation of $L*$:

$$L^* = \begin{cases} 116 \cdot \left(\frac{Y}{Y_n}\right)^{1/3} - 16 & \text{if } \frac{Y}{Y_n} > (\frac{6}{29})^3 \\ 903.3 \cdot \frac{Y}{Y_n} & \text{if } \frac{Y}{Y_n} \leq (\frac{6}{29})^3 \end{cases} \text{[5]} \tag{2.7}$$

Where Y_n represents the Y tristimulus value of the reference white point ($D65$ in our case). The $u*$ and $v*$ coordinates are then computed as:

$$u^* = 13L^*(u' - u'_n) \tag{2.8}$$

$$v^* = 13L^*(v' - v'_n) \tag{2.9}$$

Where u' and v' are derived from the XYZ tristimulus values:

$$u' = \frac{4X}{X + 15Y + 3Z} \tag{2.10}$$

$$v' = \frac{9Y}{X + 15Y + 3Z} \tag{2.11}$$

The variables u'_n and v'_n represent these same calculations performed on the reference white point. This mathematical framework ensures that equal distances in the CIELUV colour space correspond to roughly equal perceived differences in colour, addressing a key limitation of RGB-based spaces. For instance, a colour difference of $\Delta E = 1$ in CIELUV space approximates the **just-noticeable difference (JND)** threshold for human perception [5].

The practical significance of perceptually uniform spaces extends beyond theoretical colour science. In computer graphics and digital imaging, these spaces enable more intuitive colour manipulation and more accurate colour difference calculations. When performing operations such as colour interpolation or generating colour palettes, working in a perceptually uniform space helps ensure that the results align with human visual expectations, making them particularly valuable for applications in user interface design, data visualization, and digital art [15].

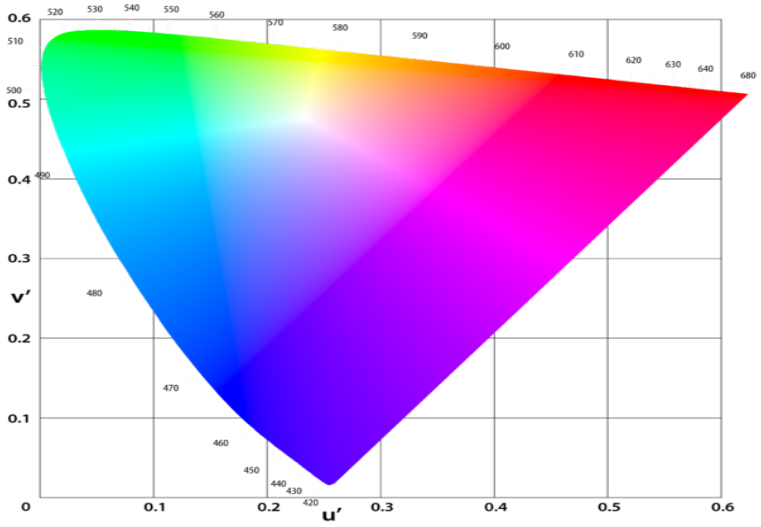

Figure 2.3: CIE1976 u' v' Uniform Chromaticity Scale [11]

This intersection of mathematical precision with perceptual accuracy represents the culmination of our exploration of colour spaces, demonstrating how a theoretical understanding of human vision can be translated into practical tools for digital colour manipulation.

2.3.2 Uniform Uniformity: Colour Appearance Models

The development of perceptually uniform colour spaces has seen significant evolution since the introduction of CIELUV. While CIELUV provided a foundation for perceptual uniformity, research by Takamura and Kobayashi demonstrated that its uniformity could be substantially improved through targeted modifications. Their work achieved a 24% improvement in uniformity by optimizing CIELUV's coefficient set while maintaining its core formulation, and a further 38% improvement through minor formulation adjustments [25]. Notably, they achieved an additional 3-5% improvement by specifically optimizing for the sRGB gamut, acknowledging the growing prevalence of sRGB-compliant content in digital applications.

The pursuit of enhanced perceptual uniformity led to the development of CIECAM02 [3], and most recently, CIECAM16 [27], which introduces a more comprehensive model of human colour perception. However, this improved accuracy came with increased computational complexity. These **colour appearance models (CAMs)** rely on trigonometric

functions which make them substantially more expensive to compute in graphics applications. The forward model for CIECAM02 requires trigonometric calculations for each colour conversion, including:

$$h = tan^{-1}(b/a)[22] \tag{2.12}$$

Where h represents the hue angle, and a and b are chromatic responses. This computational overhead becomes particularly significant in real-time graphics applications where millions of colour calculations may need to be performed per frame.

The trade-off between perceptual accuracy and computational efficiency highlights a fundamental challenge in colour space design in graphics. While CAMs offer superior perceptual uniformity compared to CIELUV, their computational demands often make them impractical for performance-critical applications. Although I will not be making use of CAMs in this paper (perhaps better for **Physically Based Rendering Pipelines** [4]), I believe that it is still important to discuss modern colour theory as CIECAM16 currently acts as the ground truth colour model of our reality (for now!).

Now that we have an introductory understanding of colorimetry, we can finally begin discussing the topic of rendering colour spaces in three dimensions!

Chapter 3

Visualizing Perceptual Colour Spaces Using Modern Rendering Practices

The topic of rendering colour spaces in three dimensions is surprisingly under-discussed. As a result of this, the available tooling for exploring our colour spaces in an interactive and *responsive* manner is restricted to very few choices. It is important for us to briefly investigate current tools before we begin working towards a modern rendering solution. Once we've identified issues with existing solutions, we'll seek to theorize a modern solution for efficiently rendering colour spaces in real-time. While implementation is outside the scope of this project, I still felt that it would be appropriate to attach some working proof of concept. In this paper, I will present a modern and scalable colour space rendering solution that makes use of WebGPU's point primitives for efficient rendering. This solution can be generalized across domains for point-cloud rendering using GPU acceleration. The goal is to finally provide a cross-platform open-source rendering solution that can be further developed and adapted for open and optimized scientific visualization tools.

3.1 Existing Solutions

Our current options seem to be *MathWorks'* MATLAB [19], GamutVision [12], and a plugin named Color Inspector 3D [1] for the scientific image processing suite ImageJ Fiji [26]. We won't be inspecting GamutVision as it suffers from issues present in the other solutions, and it hasn't received any updates since 2008 [12]. The unfortunate reality is that these products, while they accomplish their goals in displaying colour spaces in three

dimensions, do not do a very good job when it comes to fully depicting all colours at our disposal in 24-bit colour and 30-bit colour.

It is also important to mention that we can use Python libraries such as PyVista for real-time point cloud rendering, however, I found that it was quite slow with large amounts of particles.

3.1.1 Color Inspector 3D

Color Inspector 3D, as a plugin for ImageJ Fiji, represents one of the earliest attempts at interactive 3D colour space visualization. The plugin utilizes Java3D for rendering, which presents several significant limitations in modern computing environments. The plugin offers visualization capabilities for various colour spaces, but suffers from several technical limitations:

1. Java3D has lacked *official* support for well over a decade. There exists a community branch, but why choose to do graphics programming in Java when we can build something new?

2. Limited to 8-bit colour depth per channel, making it unsuitable for HDR colour space visualization.

3. Viewport resolution is very odd, making the visual output seem incorrect even if it is not (see 3.1). A proper visual will be shown later in our implementation.

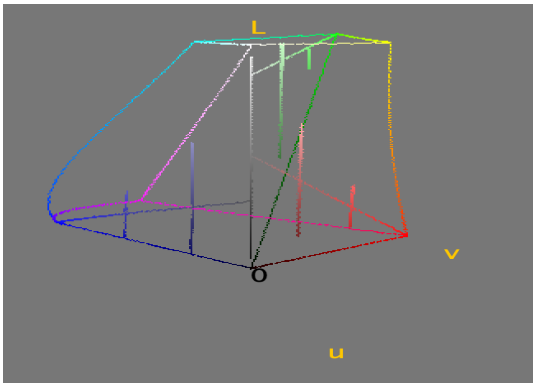

Figure 3.1: CIELUV subset rendered with a squashed shape in Color Inspector 3D

While Colour Inspector 3D served its purpose for basic colour space analysis, its architecture makes it unsuitable for modern colour science applications, particularly in the context of wide colour gamut and HDR workflows [7].

3.1.2 MATLAB

MATLAB is industry standard, heavily funded scientific computation software that offers both **vertex-based mesh rendering** and **point cloud rendering** for colour space visualization (See 3.2). A glaring problem with MATLAB is that its renderer is currently stuck using OpenGL 4.0 and WebGL [20], which means it is incapable of utilizing OpenGL 4.3 compute shaders [6] with its rendering backends. Upgrading to OpenGL 4.3 presents an issue for MATLAB as this will break continuity with their WebGL implementation which cannot support compute shaders [28]. As for MATLAB's point clouds, they do support CUDA acceleration, but this restricts usage to NVIDIA hardware with MATLAB installed. By most standards, MATLAB offers enough tools for colour space visualization, though its cost compared to the other available options makes MATLAB a difficult value proposition.

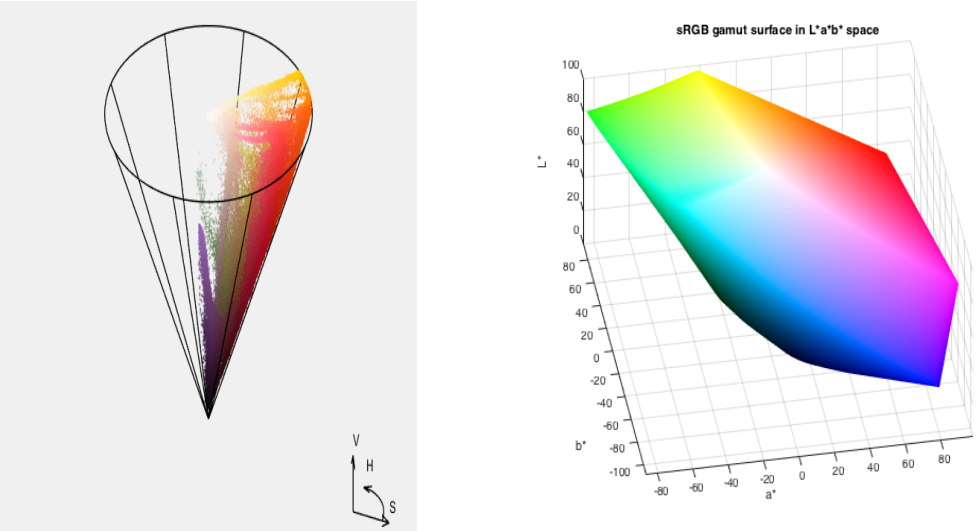

Figure 3.2: MATLAB point cloud (left) & MATLAB trisurf mesh (right) [8]

3.2 Implementation Considerations

A common thread among existing solutions is their reliance on old or deprecated graphics APIs and rendering techniques or reliance on specific hardware. This presents several challenges for modern colour space visualization:

- **Memory Management:** Traditional vertex-based approaches to colour space visualization require storing complete mesh or point cloud data in system memory, limiting the resolution to examine colour spaces. Using modern GPU storage buffers through APIs such as WebGPU, DirectX, Vulkan, and Metal, will allow for easily manipulable data on the GPU that will rarely (if ever) have to be written back to the CPU.
- **Culling:** When rendering dense colour spaces, it is important to eliminate anything that is not visible to the camera. This can save on frame time, especially when working with more than 16 million densely packed vertices in real-time.
- **Real-time Interaction:** The lack of compute shader support in older graphics APIs significantly impacts the ability to perform real-time colour space transformations and analysis. The investigation of individual points within the point cloud should be smooth.
- **Colour Precision:** Non-MATLAB tools are limited to 8-bit colour depth, making them inadequate for professional colour management workflows that require higher precision.
- **Image Loading:** Not as much as a challenge, but more of a consideration. Existing tools offer the ability to load images such as **PPM** files into 3-dimensional colour space representations for colour palette inspection or analysis. It is important that we can do the same.

Memory Requirements

If we take a moment to think about how much memory will be required to store 16 million individual vertices in memory, it begins to make sense why Color Inspector 3D operates

at such a low resolution and does not support HDR. If the sRGB gamut contains 16 million unique colours that are represented using three bytes we find that $16777216 \times 3 = 50331648\ bytes = 50.33\ megabytes$. By modern standards, this isn't very much of a memory footprint at all! Unfortunately, this is not enough information to properly visualize our colour spaces in three dimensions, as well as allowing for normalized RGB values for HDR support. To do this, we will have to represent each point in our colour space with a **minimum** of 6 floats, giving our vertex buffer a stride of 24 bytes. If we revisit our calculations, we find that $16777216 \times 24 = 402653184\ bytes = 402.65\ megabytes$. Our value also doesn't account for any extra buffers of information that might be stored which would pose a problem for older software. This means that when representing a full 24-bit colour space, we will need a bare minimum of 402.65 megabytes of VRAM to store our colour spaces. If we take a look at the graphics cards from the era of Color Inspector 3D, we'll notice that this consumes nearly all memory available to the GPUs of the time except for the Nvidia GeForce 7950 GX2 which had 1GB of VRAM. Luckily for us, GPUs have come a long way and most devices have GPUs more than capable of storing these point clouds in memory.

3.3 Modern Point Cloud Renderer

Our primary goal is to develop a renderer capable of visualizing dense particle clouds that represent colour spaces. Specifically, we need to be capable of processing and displaying approximately 16 million individual particles in real time, representing every possible colour within the sRGB colour gamut. Of course, we will offer support for HDR as well, but the computing power necessary for processing more than 1 billion vertices in real time is a little much for the scope of this project. So 16 million particles will serve as our goal for benchmarking. This leads us to our key requirements.

Key Requirements

As a proof-of-concept project, the renderer must achieve:

- Real-time performance (30+ frames per second) on standard laptop hardware
- Support for both sRGB and CIELUV colour space visualization

- Capability to import custom colour palettes via PPM files

Technical Significance

Real-time visualization sets our approach apart from existing industry solutions. While many tools can visualize colour spaces, real-time interaction enables enhanced learning opportunities and scientific applications. By achieving responsive performance on consumer hardware, we make these visualization capabilities more accessible to users.

3.3.1 Graphics API

We have selected WebGPU as our graphics API foundation. This modern web-based graphics API offers several advantages over its predecessor, WebGL2:

- Modern GPU Architecture Alignment
  - WebGPU's pipeline and bundle features align closely with contemporary GPU hardware design. This architectural alignment enables more efficient GPU resource management.
- Enhanced Feature Set
  - Provides access to advanced capabilities typically found in hardware-level graphics APIs.
  - Supports the sophisticated pipeline structure necessary for our visualization requirements.
- Optimized Rendering Capabilities
  - Native support for point primitives, enabling efficient particle rendering without geometry expansion.
  - Integration of compute shaders for parallel processing of particle data, facilitating real-time updates and transformations.
  - Implementation of indirect drawing commands, allowing dynamic adjustment of render workloads without CPU overhead.

One disadvantage to WebGPU is the slow rate of adoption from browser maintainers like Apple. Thankfully they have been working to follow in Google's and Mozilla's footsteps of implementing WebGPU backends for their browsers recently [31]. With that out of the way, I still do believe that the technical capabilities of WebGPU outshine its negatives, and will soon grow to replace WebGL2.

These technical capabilities provide the foundation for both efficient colour space visualization and the potential expansion to other particle cloud applications. The combination of compute shaders for data processing and optimized rendering of primitives enables us to achieve the necessary performance while maintaining the flexibility to handle diverse visualization scenarios. Though we could go even further and implement the features mentioned such as indirect drawing, this will be considered outside the scope of this project, so we will focus on implementing efficient pipelines while making use of point primitives. This should be more than enough to prove the value of our modern rendering pipeline concerning the other available tools.

3.3.2 Building A Modern Graphics Pipeline

Modern graphics programming APIs represent a significant evolution from OpenGL/WebGL through the introduction of programmable graphics pipelines. These pipelines serve as sophisticated GPU abstractions that enable developers to define device behaviour with greater precision and control explicitly. Unlike OpenGL/WebGL's approach of using sequential function calls to establish renderer state, modern APIs emphasize defining immutable pipeline states upfront. This architectural shift brings substantial performance benefits, as pipeline definitions need only occur once—either at program initialization or object creation—significantly reducing CPU overhead. WebGPU provides access to two distinct pipeline types, each serving specific rendering needs.

WebGPU Render Pipelines

The render pipeline in our implementation follows WebGPU's standardized resource architecture, establishing a clear data flow from buffer resources to the final rendering output. Our pipeline configuration, defined in *pointclouds/pointcloud.ts*'s **initRenderPipeline()**,

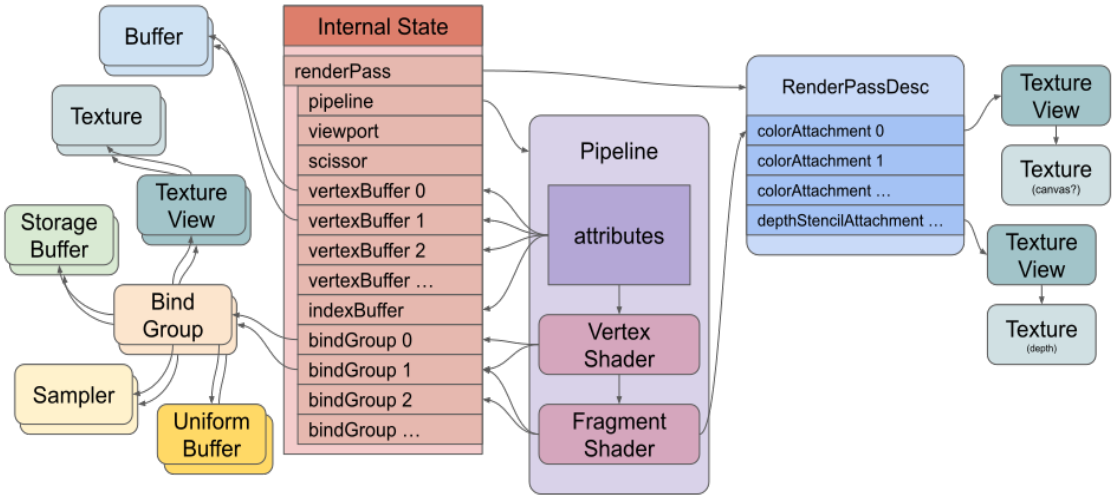

Figure 3.3: Visual Depiction of WebGPU's Complete Rendering Resource Graph [29]

orchestrates the interaction between various WebGPU resources to enable efficient particle visualization across different colour spaces.

The pipeline's data flow begins with two uniform buffers, managed through a dedicated bind group layout. These buffers maintain essential transformation matrices, connecting to the pipeline through bind group entries that are specifically visible to the vertex shader stage. This aligns with the resource graph's demonstration of how bind groups serve as intermediaries between buffer resources and the shader stages.

The vertex processing stage incorporates a structured buffer layout that mirrors the resource graph's vertex buffer configuration. Our implementation utilizes a single vertex buffer with a 24-byte stride, containing interleaved position and colour data. This buffer connects directly to the vertex shader through the pipeline's internal state, as illustrated in the resource graph's vertexBuffer entries (see fig 3.3). The vertex attributes are carefully mapped to shader locations 0 and 1, establishing the crucial link between buffer data and shader inputs.

The pipeline's fragment processing stage connects to colour attachments as shown in the resource graph's RenderPassDesc section. Our implementation configures these attachments to match the system's preferred canvas format, ensuring optimal display compatibility.

To enhance visual fidelity, our pipeline incorporates several features represented in the re-

source graph's Pipeline node. We employ point-list topology for particle rendering, depth testing for proper occlusion handling, and 4x multisampling for anti-aliasing. These configurations are managed through the pipeline's internal state, affecting how the vertex and fragment shaders process and output data. The overall architecture demonstrates how WebGPU's resource graph enables efficient particle visualization through a carefully orchestrated sequence of buffer management, shader processing, and render pass execution. This structured approach allows our implementation to maintain consistent performance while handling different color space representations within a single pipeline configuration.

WebGPU Compute Pipelines

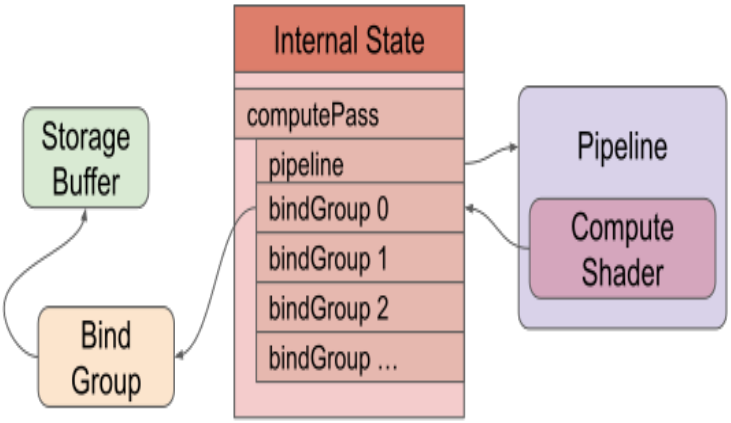

Figure 3.4: Visual Depiction of WebGPU's Compute Pipeline [29]

In our particle renderer, compute pipelines handle the complex mathematical operations required for particle position transformation and colour space transformations. These pipelines will perform the critical functions of calculating colour space conversions (e.g., sRGB to CIELUV) and handling general transformation computations for particle translation (if needed). Compute shaders execute these calculations in parallel, processing multiple particles simultaneously to achieve optimal performance.

For instance, when transitioning between different colour spaces, the compute pipeline efficiently processes the mathematical transformations for all particles in parallel, updating their positions according to the selected colour space's coordinate system. This parallel processing capability ensures smooth real-time performance even with large particle counts,

making it essential for maintaining interactive visualization rates while handling complex colour space calculations.

Our rendering system makes use of four different compute pipelines, each optimized for specific tasks in the visualization process. The primary pipeline initializes our vertex buffer as a general storage buffer, enabling the instantaneous creation of 16 million points for our point cloud directly on the GPU. This eliminates CPU overhead by avoiding data transfers between CPU and GPU memory. While a secondary pipeline exists for applying transformations to vertex data, it serves primarily as a debugging tool. The remaining two pipelines, found in **pointclouds/cieluv.ts** and **pointclouds/linearrgbcube.ts**, handle the core functionality of populating vertex and colour data in our vertex storage buffer. The colour space conversions performed in the shader kernels are simply adaptations of the equations laid out throughout **Chapter 2**,

When processing PPM textures, our system addresses GPU memory alignment requirements by treating three 8-bit colour channels as a single 32-bit unsigned integer. We add an alpha channel to ensure full utilization of the 32-bit space, allowing precise point indexing from our compute workgroups based on texture data. This capability supports advanced research applications, such as implementing depth-based point opacity, or analyzing Just Noticeable Differences (JNDs) in CIELUV subsets (see Fig 3.5).

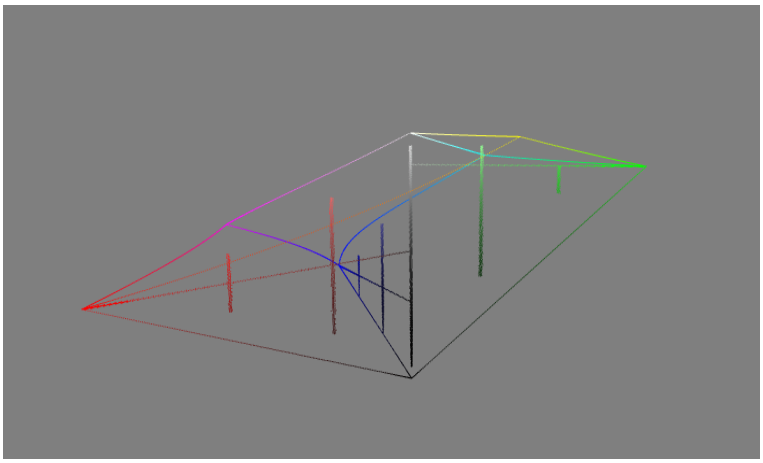

Figure 3.5: 8800 point CIELUV subset rendered using WebGPU renderer

3.3.3 Results

All code for this project can be found on the project's GitHub repository: www.github.com/klukaszek/ColourSpaces.

An interactive demo can be found at www.kylelukaszek.xyz/ColourSpaces.

A little menu was added to add interactivity to the renderer. From this menu, we can view the controls, load PPM files, or swap between colour spaces. In figures 3.6 and 3.7 we can see that the point clouds draw the appropriate colours as well as the appropriate shapes. This indicates that our rendering pipeline does indeed work as intended. Now that we know that the renderer draws the correct things to the screen, we can now review the performance!

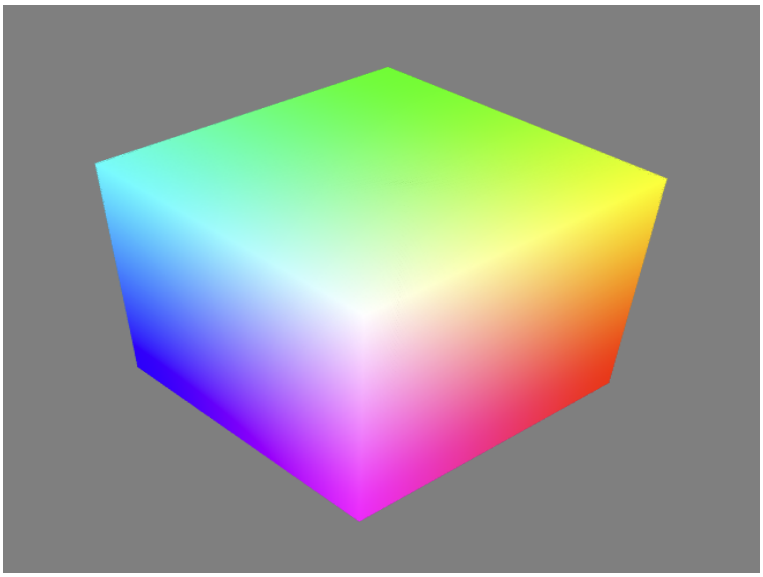

Figure 3.6: 16 million point cloud of sRGB cube using WebGPU renderer

3.3.4 Performance

Evaluating performance is important for us as we are attempting to provide a real-time tool. Most real-time tools target 33ms response time so that we can achieve 30fps. If possible, we are looking to get much higher than that though since 30fps is not even enough to match the refresh rate of most modern displays.

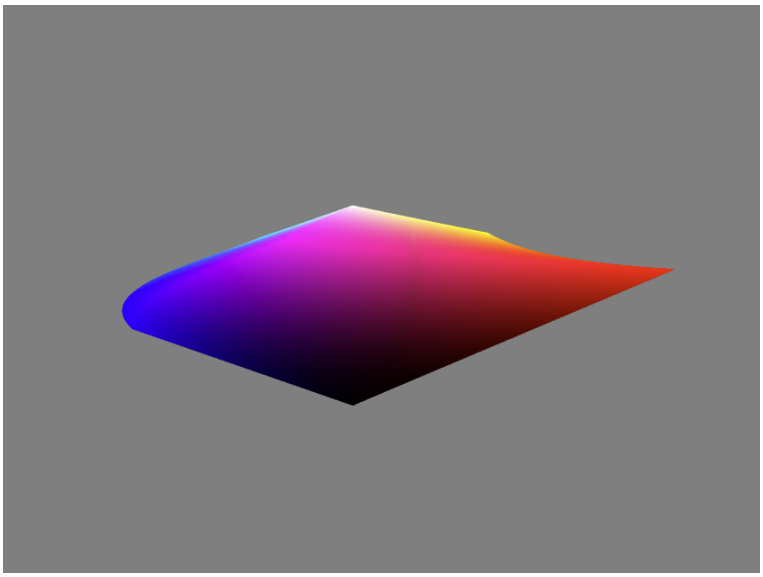

Figure 3.7: 16 million point cloud of CIELUV using WebGPU renderer

Memory

The resulting browser tab running our new renderer consumes anywhere between $30-70mb$ of RAM while our scene is loaded. This is great as we know for a fact that all of our data exists on the GPU and that no CPU overhead is being incurred.

Frametime/Framerate

I will be testing the frametime of the renderer in 4 different states on two different systems to evaluate frametime. The first system is my desktop PC which has an Intel 9600K CPU, and an NVIDIA RTX 2070 Super GPU, using Google's Dawn Backend. The second system will be a base model Apple MacBook Air equipped with an M3 chip. These selections were made with consumer hardware in mind. We will be measuring average frametime over 30 seconds. Our target frametime is $< 33ms$ for a smooth 30 fps.

- **Test Case 1:** Stand still with the colour space rotating around the scene.
- **Test Case 2:** Pause the rotation of the colour space and evaluate performance with no translations being applied via compute shaders.
- **Test Case 3:** Stand within the colour space. See 3.8.
- **Test Case 4:** A PPM texture in place of a full resolution point cloud.

Test Case	Desktop PC (Intel 9600K + RTX 2070 Super)	MacBook Air (Apple M3)
1. Rotating Color Space	6.99 ms	25.64 ms
2. Static Color Space	6.99 ms	20.83 ms
3. Interior View	11.34 ms	16.666 ms
4. PPM Texture	6.99 ms	16.666 ms

Table 3.1: Frametime Comparison Across Test Systems

Based on table 3.1, we can see that **both platforms consistently target < 33ms frametime**. This means that we have achieved our goal of implementing a real-time particle renderer by just making use of compute pipelines and GPU instancing! Interestingly enough the MacBook performs better when travelling inside of the colour space while my PC struggles most when approaching the point cloud. This goes to show the rapid advancements in GPU technology that Apple has made with their M series chips for base-level systems.

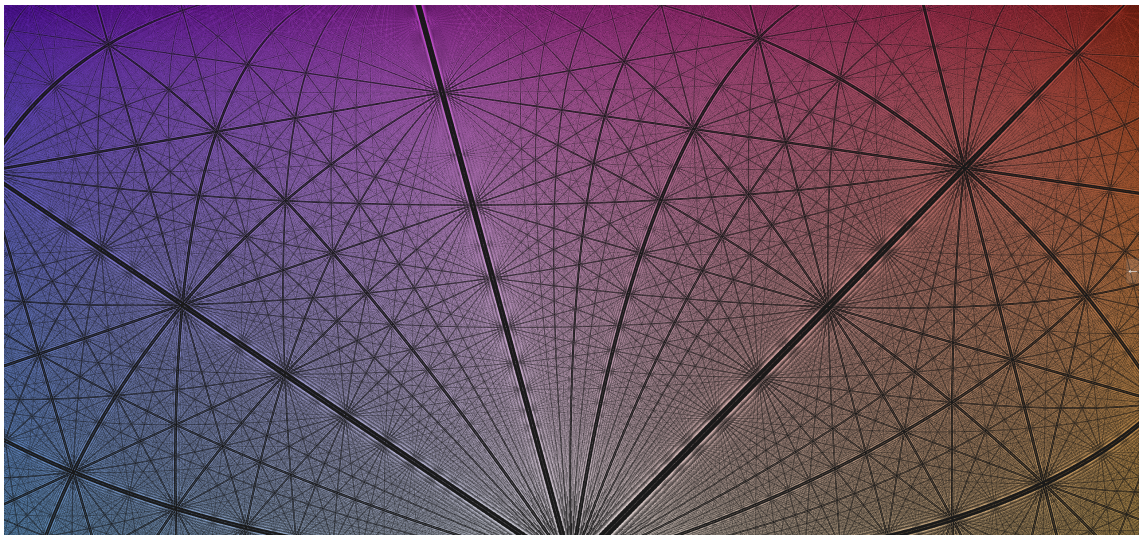

Figure 3.8: Interior view of CIELUV colour space

3.3.5 Future Work

- **Tonemapping:** Modern rendering pipelines offer support for HDR tonemapping. WebGPU offers HDR tonemapping hidden behind a flag. We will not be implementing HDR tonemapping as I currently lack any means to verify that it works, but the option is there, and there is a tutorial written in Rust that outlines the implement-

ation process available on *learnwgpu* [13]. There is also HTML canvas tonemapping which should also offer the expected results as well.

- **Extended Image Support:** Being able to load PNGs, JPEGs, HDRIs, etc., would be an improvement of our current implementation that only supports PPM files. Extended file type support could add more opportunities for interesting visualizations in three dimensions, HDRIs especially.

- **Volumetric Rendering:** Making use of ray-marching in addition to **kd-tree** data structures could prove useful for greatly optimizing performance and developing a volumetric rendering pipeline similar to those used in CT scanning. Density-based rendering could offer a large leap in performance as millions of points are being tested on each draw that end up occluded, or simply imperceptible due to JND.

- **Continuous Level of Detail:** Schutz et al. [24], present a method for continuous level-of-detail (CLOD) rendering of point clouds that could offer significant performance increases. By implementing their approach of using a compute shader to dynamically select points based on target spacing and randomizing point LODs, we could achieve smoother transitions between detail levels and reduce artifacts present in our current implementation, especially beneficial for possible VR applications for participant research [24]. Additionally, their technique of gradually increasing the size of newly visible points could enhance the visual quality and user experience of the renderer, particularly when navigating through large point cloud datasets. This would require a transition away from instanced point primitives and instead using SDF-drawn points on instanced quads.

- **Splatting:** In September 2024, Yueyu Hu et al. [14] presented a novel approach for point cloud rendering using the increasingly popular Gaussian splatting approach. The paper's neural approach to estimating 3D elliptical Gaussians could enhance our WebGPU renderer by enabling high-quality surface reconstruction without per-scene optimization. Their P2ENet architecture demonstrates impressive generalizability while maintaining sub-30ms preprocessing latency, making it suitable for real-time VR/AR applications. By analyzing both local and global point cloud structure, the method optimizes elliptical parameters and surface normals of Gaussians to improve

rendering quality, particularly for sparse or noisy regions [14]. The differentiable splatting technique achieves over 100 FPS post-preprocessing. Most importantly, this modern architecture enables high-quality splatting without compute-intensive surface reconstruction while maintaining smooth textures and accurate surface normals.

Chapter 4

Conclusions

The development and implementation of our WebGPU-based colour space visualization system demonstrates that modern web technologies can deliver professional-grade rendering capabilities without specialized hardware requirements. Our implementation achieves consistent frame times below 30 milliseconds on consumer hardware, establishing real-time performance even without extensive optimization efforts. This performance threshold was achieved primarily through architectural decisions, specifically the adoption of compute shaders and point-based rendering techniques, rather than through complex optimization strategies. The success of this implementation carries significant implications for both educational and professional applications in the field of colour science. By achieving responsive performance through web browsers, we have effectively lowered the barrier to entry for sophisticated colour space visualization. The combination of accessibility and performance validates our architectural approach and suggests that further optimization could yield even more impressive results.

This work demonstrates that as the adoption of WebGPU continues to increase, we will soon be capable of handling computationally intensive visualization tasks while maintaining the accessibility and convenience of web-based applications. As web technologies continue to evolve, our implementation serves as a proof of concept for future developments in browser-based colour science visualization tools. The achievement of sub-33ms frame times on modern consumer hardware, coupled with the potential for additional optimization, positions this solution as a viable alternative to traditional desktop applications for colour space visualization and analysis.

References

3D Color Inspector/Color Histogram — imagej.net (n.d.). https://imagej.net/ij/plugins/color-inspector.html (cit. on p. 14).

A Standard Default Color Space for the Internet - sRGB — w3.org (n.d.). https://www.w3.org/Graphics/Color/sRGB.html. [Accessed 13-11-2024] (cit. on p. 8).

al., Li et (2004). *A colour appearance model for colour management systems: CIECAM02*. CIE. ISBN: 978 3 901906 29 9 (cit. on p. 12).

Bratuž, Nika, Helena Gabrijelčič Tomc and Dejana Javoršek (June 2017). 'CIECAM02 and Perception of Colour in 3D Computer Generated Graphics'. In: *TEKSTILEC* 60.2, pp. 97–106. ISSN: 2350-3696. DOI: 10.14502/tekstilec2017.60.97-106. URL: http://dx.doi.org/10.14502/TEKSTILEC2017.60.97-106 (cit. on p. 13).

Colorimetry 3rd Edition (2004). en (cit. on pp. 4, 9, 11).

Compute Shader - OpenGL Wiki — khronos.org (n.d.). https://www.khronos.org/opengl/wiki/Compute_Shader (cit. on p. 16).

Display Modes — home2.htw-berlin.de (n.d.). https://home2.htw-berlin.de/~barthel/ImageJ/ColorInspector/HTMLHelp/Darstellungsmodi.htm (cit. on p. 16).

Eddins, Steve (n.d.). *Displaying a Color Gamut Surface*. https://blogs.mathworks.com/steve/2015/04/03/displaying-a-color-gamut-surface/ (cit. on p. 16).

File:CIExy1931.png - Wikimedia Commons — commons.wikimedia.org (n.d.). https://commons.wikimedia.org/wiki/File:CIExy1931.png. [Accessed 11-12-2024] (cit. on p. 5).

File:RGB farbwuerfel.jpg - Wikimedia Commons — commons.wikimedia.org (n.d.). https://commons.wikimedia.org/wiki/File:RGB_farbwuerfel.jpg. [Accessed 11-11-2024] (cit. on p. 7).

File:SRGB chromaticity CIE1931.svg - Wikimedia Commons — commons.wikimedia.org (n.d.). https://commons.wikimedia.org/wiki/File:SRGB_chromaticity_CIE1931.svg. [Accessed 13-12-2024] (cit. on pp. 9, 12).

Gamutvision - explore color spaces, gamut mappings, and rendering intents — gamutvision.com (n.d.). http://www.gamutvision.com/ (cit. on p. 14).

High Dynamic Range Rendering | Learn Wgpu — sotrh.github.io (n.d.). https://sotrh.github.io/learn-wgpu/intermediate/tutorial13-hdr/#what-is-high-dynamic-range. [Accessed 14-12-2024] (cit. on p. 27).

Hu, Yueyu et al. (2024). *Low Latency Point Cloud Rendering with Learned Splatting*. DOI: 10.48550/ARXIV.2409.16504. URL: https://arxiv.org/abs/2409.16504 (cit. on pp. 27, 28).

Hunt, R W G (Sept. 2004). *The reproduction of colour*. en. 6th ed. The Wiley-IS&T Series in Imaging Science and Technology. Hoboken, NJ: Wiley-Blackwell (cit. on pp. 10, 11).

Hunt, R. W. G. and Michael Pointer (2011). *Measuring colour*. Wiley (cit. on pp. 2–6).

ICC (n.d.). *sRGB — color.org*. https://www.color.org/chardata/rgb/srgb.xalter. [Accessed 13-11-2024] (cit. on p. 8).

Kuehni, Rolf G. (2016). 'How many object colors can we distinguish?' In: *Color Research & Application* 41.5, pp. 439–444. DOI: https://doi.org/10.1002/col.21980. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/col.21980. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/col.21980 (cit. on p. 1).

MathWorks (n.d.). https://www.mathworks.com/products/matlab.html (cit. on p. 14).

MATLAB (n.d.). *MATLAB rendererinfo*. https://www.mathworks.com/help/matlab/ref/rendererinfo.html. [Accessed 11-11-2024] (cit. on pp. 1, 16).

Meessen, A. (Nov. 1967). 'A simple non-linear theory of color perception and contrast effects'. In: *Kybernetik* 4.2, pp. 48–54. ISSN: 0023-5946. DOI: 10.1007/bf00292171. URL: http://dx.doi.org/10.1007/BF00292171 (cit. on p. 8).

Moroney, Nathan et al. (Jan. 2002). 'The CIECAM02 color appearance model'. In: vol. 10, pp. 23–27 (cit. on p. 13).

Nassau, Kurt (Oct. 2024). *Colour*. URL: https://www.britannica.com/science/color (cit. on p. 2).

Schutz, Markus, Katharina Krosl and Michael Wimmer (Mar. 2019). 'Real-Time Continuous Level of Detail Rendering of Point Clouds'. In: *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. IEEE, pp. 103–110. DOI: 10.1109/vr.2019.8798284. URL: http://dx.doi.org/10.1109/VR.2019.8798284 (cit. on p. 27).

Takamura, S. and N. Kobayashi (n.d.). 'Practical extension to CIELUV color space to improve uniformity'. In: *Proceedings. International Conference on Image Processing*. Vol. 2. ICIP-02. IEEE, II–393–II–396. DOI: 10.1109/icip.2002.1039970. URL: http://dx.doi.org/10.1109/ICIP.2002.1039970 (cit. on p. 12).

Team, The Fiji (n.d.). *Fiji: ImageJ, with "Batteries Included" — fiji.sc*. https://fiji.sc/ (cit. on p. 14).

'The CIE 2016 Colour Appearance Model for Colour Management Systems: CIECAM16' (n.d.). In: (). DOI: 10.25039/tr.248.2022. URL: http://dx.doi.org/10.25039/TR.248.2022 (cit. on p. 12).

WebGL 2.0 Specification — registry.khronos.org (n.d.). https://registry.khronos.org/webgl/specs/latest/2.0/ (cit. on p. 16).

WebGPUFundamentals (n.d.). *WebGPU Draw Depencies*. https://webgpufundamentals.org/webgpu/lessons/resources/webgpu-draw-diagram.svg (cit. on pp. 21, 22).

Webster, Michael A. and Deanne Leonard (Oct. 2008). 'Adaptation and perceptual norms in color vision'. In: *Journal of the Optical Society of America A* 25.11, p. 2817. ISSN: 1520-8532. DOI: 10.1364/josaa.25.002817. URL: http://dx.doi.org/10.1364/josaa.25.002817 (cit. on p. 10).

Wyrzykowski, Mike (n.d.). *WebGPU now available for testing in Safari Technology Preview*. https://webkit.org/blog/14879/webgpu-now-available-for-testing-in-safari-technology-preview/. [Accessed 14-12-2024] (cit. on p. 20).