

ALLEGHENY COLLEGE
DEPARTMENT OF COMPUTER SCIENCE

Senior Thesis

Real-Time Non-photorealistic Color Quantization

by

Maximillian Mandel

ALLEGHENY COLLEGE

DEPARTMENT OF COMPUTER AND
INFORMATION SCIENCE

Project Supervisor: **Gregory M. Kapfhammer**
Co-Supervisor: **Douglas Luman**

3 May 2024

Abstract

A non-photorealistic rendering effect changes an image render through the use of various graphics post-processing algorithms. This is done with the goal of stylizing an image in order to make it appear more visually interesting, diverse, or to highlight specific elements within an image. This thesis describes in detail a series of graphics processing algorithms intended to be ran sequentially in a post-processing pipeline. The proposed stylization of images in real time uses techniques such as edge detection, ordered dithering, and color quantization. This effect, titled real-time color quantization, allows existing renders to take on a new and unique look with the intent of replacing or modifying the core art-style of a real-time application. This thesis focuses on analyzing and developing algorithms that can generate this stylistic effect while being suitable for various real-time applications, leveraging existing knowledge in the field and existing graphics processing techniques.

Various individual algorithms and their logics and implementations are proposed. In addition to this, a Unity project has been developed that demonstrates each of the proposed algorithms effects both individually and with each other. By implementing these algorithms in a Unity environment, this thesis provides both a proof of concept and a framework example for how these algorithms might be implemented into an existing project. This Unity demo is used to run various experiments on the algorithms, testing the impact they have on performance in general and determine whether they are suitable for real time. Generally, the proposed algorithms have a noticeable performance impact on the application, the significance of which varies widely depending on the specific device being used to run the application. In general, more powerful devices that are not limited by the CPU will experience a more significant reduction in performance, although the experiments indicated that the performance will never drop below what is considered to be real time.

Table of Contents

Introduction	5
Motivation	5
Current State of the Art	6
Goals of the Project	9
Ethical Implications	10
Impact	11
Related work	12
Depth-of-Field & Edge Detection	12
Blue Noise Dithering	14
Ordered Dithering	16
Color Spaces	17
Where This Project Stands	18
Method of approach	19
Algorithm Overview	19
Edge Detection	20
Color Quantization	22
Dithering	25
Blue Noise Dithering	26
Ordered Dithering	27
Processing Pipeline	28
Running the Demo	31
How to Run	32
Modifying the Demo	32
Experiments	33
Experimental Design	33

Evaluation	34
Analysis	38
Threats to Validity	39
Conclusion	41
Summary of Results	41
Future Work	42
Future Ethical Implications and Recommendations	43
Conclusions	44
References	46

List of Figures

1	The Legend of Zelda: Breath of the Wild	7
2	Return of the Obra Dinn	8
3	Layered non-photorealistic rendering with anisotropic depth-of-field filtering example	13
4	Progressive image stippling and greedy blue noise importance sampling example	15
5	Depth Texture	21
6	Depth Edge Detection	22
7	Color Quantization	25
8	Top-left: No Post-processing, Top-right: Blue Noise Dithering, Bottom-left: Close Blue Noise Dithering, Bottom-Right: Blue Noise Texture	27
9	Top-Left: No Post-processing, Top-right: Edge Detection, Bottom-left: Color Quantization, Bottom-right: Color Quantization with Dither	29
10	Graphics Pipeline Flowchart	30
11	Top-Left: 12 Color Palette, Top-right: 4 Color Palette with DoF, Bottom-left: 16 Color Palette with DoF, Bottom-right: 4 Color Palette	31
12	Mean Frames-Per-Second	36

List of Tables

1	Performance Metrics for Each Algorithm Independently Over 30 Seconds Each. (Desktop)	34
2	Performance Metrics for the Fully Configured Processing Pipelines Over 30 Seconds Each. (Desktop)	35
3	Performance Metrics for Each Algorithm Independently Over 30 Seconds Each. (Laptop)	35
4	Performance Metrics for the Fully Configured Processing Pipelines Over 30 Seconds Each. (Laptop)	35

Introduction

Motivation

Most graphics programming is done with the goal of making an image look more realistic. This is achieved through the use of various techniques that have been developed over the years, such as algorithms that reduce the amount of noise in an image through artificial blending. In real-time applications, such as video games, this is especially true. This is because it enables developers to add post-processing effects like depth of field that can cheaply increase the visual fidelity of the final product, thus improving the end-user experience.

However, there is another, smaller use for graphics programming techniques that is rarely used in real-time environments. As described by Simon Schofield in his thesis *Non-photorealistic Rendering: A critical examination and proposed system*, Non-photorealistic rendering is “any type of computer graphic imagery which tries to get away from the paradigm of photorealism.” [9]. A non-photorealistic rendering (NPR) algorithm performs the act of artificially transforming an image with the goal of applying a specific style or effect to it. This stylized version of the image is often used to show extra detail present in the image that is normally overshadowed. Alternatively, it can also be used to create detail where there is none. Non-photorealistic rendering can be used to generate a single, consistent art style across many different images. When these NPR techniques are introduced in real-time software such as a video game, they can be used to drastically change the art style of the media without needing to directly modify any of the assets on a fundamental level. [5]

This has the result of greatly changing the style and visuals within a piece of real-time media in a consistent and reproducible way. The immediate benefit of performing this visual transformation is that developers could create

an effect that is visually interesting without needing to sink a large amount of time or money into creating or remaking digital assets. [5]. Furthermore, the developer could use these techniques to react to the media in real-time, giving them another dimension of freedom when designing, implementing and shipping the final product.

Current State of the Art

In the current age of real-time application development, it is common for programmers to design their applications, such as games with shaders, programs run on the GPU, that allow for advanced post-processing effects. [3]. This is especially true among AA and AAA game development studios as shaders provide a cheap, time-efficient method to increase the visual fidelity of a game. [8]. That being said, these shaders are rarely ever a primary focus in the games' art direction, and are simply developed to help augment the pre-existing visuals. One of the most common examples of shaders being used to alter a game's visuals is with cel-shaded graphics. Nowadays, quite a few games use cel shading in order to create a unique, cartoon-esque look that avoids compromising a game's visual fidelity. One of the most notable examples of this is Nintendo's *The Legend of Zelda: Breath of the Wild*.



Figure 1: The Legend of Zelda: Breath of the Wild

Here, cel shading is used as a part of a larger art style in order to achieve a specific design goal. However, while cel-shading certainly fulfills the definition of a non-photorealistic rendering based shader, there are some rare cases of a more aggressive approach being attempted in recent years.

A game utilizing non-photorealistic rendering shaders as a primary component of the game art is Lucas Pope's *Return of the Obra Dinn*.

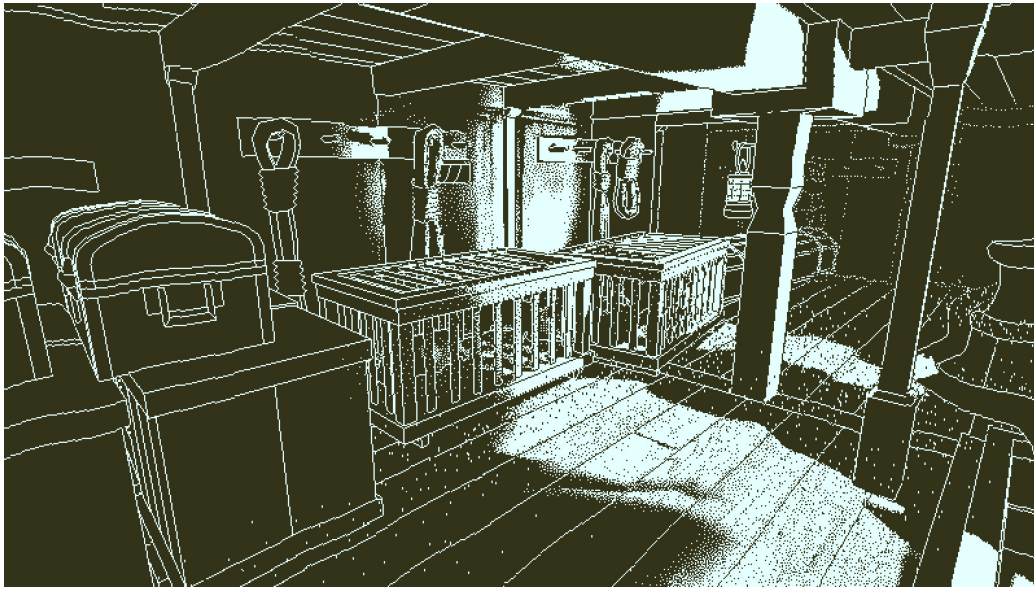


Figure 2: Return of the Obra Dinn

This game utilizes a few different techniques in order to create its unique visuals. The simpler effects involved include both an edge detector used to outline objects, as well as a dithering effect used to represent and disperse some of the lighting within the scene. [10]. By using these techniques, the game is able to look visual interesting without needing to change each individual asset to match a specific art style. Furthermore, the game is able to make use of assets that may normally be considered too low resolution or geometrically simple, which enables the developers to save costs both in terms of in-game performance and development. These techniques ultimately culminate in a diverse style that is easy to recognize and contains its own unique identity.

While there are certainly some software developers in the world currently exploring and utilizing NPR shaders for real-time applications, [5] the field is still quite young and there has yet to be substantial development and innovation. Applications that take a similar approach to Lucas Pope’s *Return of the Obra Dinn* are few and far between. Because these products are largely made by independent software developers, there is a large cost to learn how to develop and implement these techniques. This is especially true if you wanted to create an effect that is unique from other products on the market, and not

just a carbon copy of the few examples available. Because of these technical drawbacks, most developers instead opt to spend more resources investing into the art style of the application through the creation of additional assets.

Goals of the Project

With these potential benefits and limitations in mind, this project aims to help developers further develop art and visuals in their real-time applications. It does this through its development of non-photorealistic rendering algorithms that can be implemented into a variety of real-time environments. This project uses existing knowledge in the fields of graphics programming and image post-processing in order to innovate and expand what is possible through the use of shaders and other graphical programs. By thoroughly dissecting each of the core ideas and motivations behind each algorithm and its implementation, this project will be able to be used by many external developers to create new and interesting visual effects through the use of non-photorealistic rendering, further reducing the cost of visual real-time application development.

Specifically, this project will implement multiple distinct image processing algorithms into a consolidated post-processing pipeline. These algorithms include techniques such as dithering, color quantization, edge detection, and depth-of-field, among others. By integrating these algorithms into a single image processing effect, this project aims to develop a real-time, non-photorealistic rendering effect that can be used to augment and modify the visuals of nearly any real-time application. In doing so, this thesis will help develop the field of NPR in a real-time context, further enabling software developers to save resources when developing suitable applications. This effect and its implementations are described in full at a fundamental level, allowing developers to use and modify it as they see fit. This will allow the effect to be integrated into any computer-graphics related system, enabling it to be integrated into many different types of application development environments. Furthermore, this application will also prove the flexibility of these algorithms, showcasing the light-weight and portable nature of them by providing an example taking the form of a demonstration.

Accompanying these algorithms and their explanations is a short demo that was developed using the Unity Game Development Engine. The demo takes the form of a very short indie exploration game that has the player

exploring a dynamic environment while solving simple puzzles. Each of the scenes within the demo are built using only the basic geometry Unity provides, while the rest of the art is generated purely through the use of shaders and their graphical effects. This demo showcases the algorithms and their implementations in a dynamic environment, which will not only show the diversity of what is possible with them, but also prove their flexibility. By including this demo, the project aims to provide a proof of concept that can help developers decide whether or not these non-photorealistic rendering effects would be suitable for their own real-time applications. Furthermore, this demo provides a framework that developers can use to grow accustomed with the idea of a non-photorealistic based graphics pipeline.

By providing thorough documentation and detail as to how the image-processing effect and demonstration run, this project hopes to enable more software engineers to use non-photorealistic rendering algorithms to develop interesting looking, yet clear, graphics-based real-time applications. This will not only save smaller development studios resources, but also allow real-time applications a flexible appearance in their art-styles that would not ordinarily be possible with traditional methods. Namely, these techniques would allow developers to fine-tune the artistic appearance of the application during runtime in response to a users input, a feat which is normally not possible using static assets. Through the reduced cost of development and the freedom provided by these algorithms, developers should be given much more freedom in how they approach the art-style of a real-time project.

Ethical Implications

From an ethical standpoint, there are a couple minor points that could stand out as potential problems. One of these would be the potential for the NPR effects to, at least partially, replace standard creative asset development in a software development setting. [7]. If a company would be so inclined, they could, in theory, replace some or all of the main artists driving a product's art direction with these graphical effects. This would save costs on the project on paper and could lead to many jobs being lost. However, this problem has yet to prove realistic at a large scale. While these algorithms are certainly powerful and able to save quite a bit of time in some instances, it is simply not possible for them to recreate the visuals that a dedicated team of artists and designers could create. This is especially true in the case of applications that

attempt to replicate photorealism, as detailed models and advanced lighting effects are simply too important to forgo in favor of a post-processing effect.

One last ethical issue to discuss is the problem of accessibility. This is an issue when talking about graphics processing because the effects are generally expected to be generated on the end-user's device. What this means is that the program this algorithm would be built into must have specific hardware available, namely a graphics card that meets the shader's requirements. That being said, almost every modern device created within the last 15 years has some form of graphics processing available to drive the already demanding visuals of operating systems' UI. Furthermore, since this NPR effect has a deliberate focus on maintaining real-time feasibility, it is highly efficient and should have almost no effect on a consumer's ability to run a product.

Impact

This thesis furthers the research and development of non-photorealistic rendering algorithms to be used in real-time applications such as video games. It does this through the development of new graphics post-processing algorithms. These algorithms are developed to be highly flexible and extendable so that they might be used in both existing and new projects. Furthermore, although a complete post-processing stack is proposed as an example of how a developer might use these algorithms, the algorithms themselves are not intended to be used in any single specific way. Rather, the developer has full control to take the algorithms proposed in this thesis and modify and extend them to fit specific purpose or vision at the discretion of the developer.

The remaining chapters in this thesis will analyze and identify existing examples of non-photorealistic rendering processing effects and identify which parts can be related to, modified, or replaced with new innovative effects. The thesis then describes a method for implementation which describes in detail the various algorithms used at a high level and how they might be implemented into a new or existing project. Finally, experiments are ran to determine the performance impact of running these post-processing effects, with the goal of determining and highlighting in a transparent fashion whether or not they are suitable to be used in real-time when given a specific context.

Related work

While the existence of NPR algorithms being used in video games and other real-time applications is in itself uncommon, there is still a substantial amount of research done on the development and implementation of these effects for the purposes of photographic post-processing. [5]. Furthermore, developments of digital camera filters have also played a part in real-time NPR development, leading to some new innovations in the field.

Depth-of-Field & Edge Detection

An article published in 2019 by the name of *Layered non-photorealistic rendering with anisotropic depth-of-field filtering* [4] discusses the creation and development of a non-photorealistic rendering technique that samples a depth-of-field (DoF) based rendering of a scene and then increases the relative abstraction. This article, written by Jong-Hyun Kim and Jung Lee, attempts to address an issue with the color abstraction and the subsequent edge extraction lacking a clear, defined art style that is aesthetically pleasing to the viewer. In order to solve this, the article proposes a technique that extracts an anisotropic DoF region, performs adaptive color abstraction, adaptive line extraction, and then ultimately combines the results into a completed image render. [4].

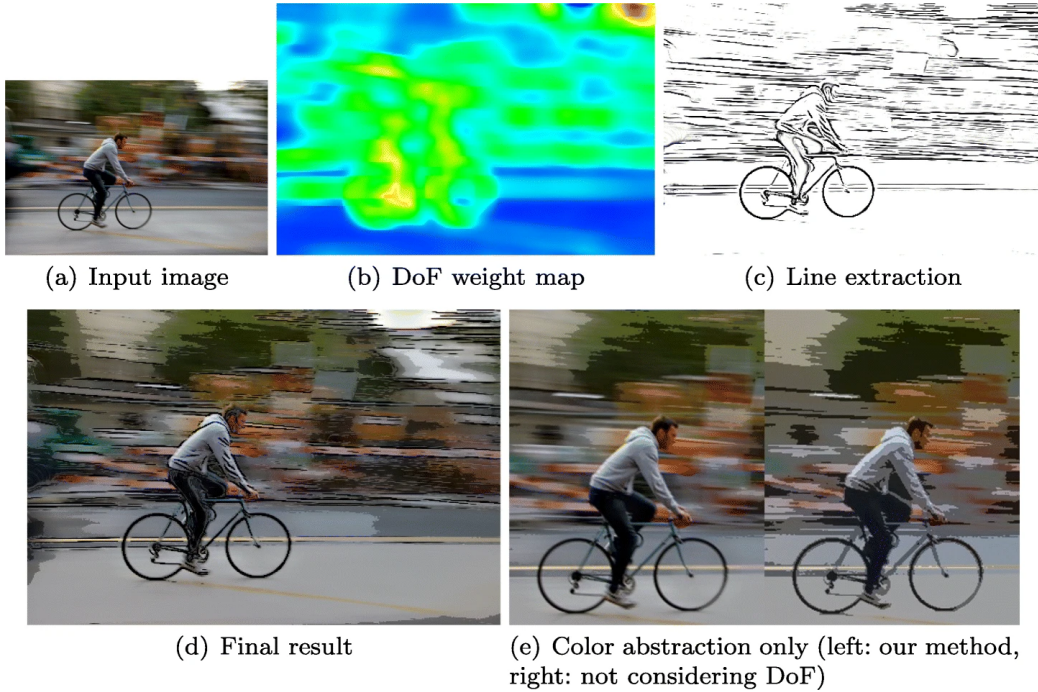


Figure 3: Layered non-photorealistic rendering with anisotropic depth-of-field filtering example

This image shows the completed effect in action, as well as a sample image taken at each of the steps towards generating the effect. Here, we can see how the image is stylized through the author’s NPR technique in order to create a render that while not photorealistic, is aesthetically pleasing.

This idea of aesthetic imagery through abstraction is a common one when discussing NPR effects. Some of the methods of reaching the visual goals discussed in the paper are also used in this project. The edge detection and line extraction step taken in this work is done through the use of a Difference of Gaussians (DoG) filter. [4]. This project experimented heavily with the use of this filter to outline and emphasize distinct objects. Although this project ultimately went with a different method for the sake of performance, the ideas behind the work are very much the same. Both techniques utilize a method to extract lines of contrast, and then layer these lines over an abstraction in the work. The end result is the ability to heavily modify and stylize a visual rendering, while still maintaining the visual clarity that may

be required in some images real-time applications.

Another similarity between Kim and Lee’s proposed technique and this project is the use of a depth-of-field algorithm. In the article, it is used as a core processing step to isolate the focus of the image and preserve some detail during the abstraction step. [4]. In contrast to this, this project’s DoF algorithm is run after the core NPR effect, and is instead used to accentuate specific details in the completed render. Furthermore, this project’s DoF algorithm works through the use of a standard Gaussian blur filter comparison that mixes two textures through real-time application specific information such as depth. Lacking this type of depth-based information, the paper needed to implement theirs through an anisotropic Dof filter that measures the difference weights of an image’s pixels. Although the algorithms are implemented differently, the end result from the depth-of-field pass is largely the same, while the uses for them are different.

Ultimately, this paper was a good baseline to compare the implementations and effects proposed by this thesis. It allowed for a direct visual comparison of different methods of edge detection. It also provided good examples for uses of depth-of-field post-processing and provided a good overview for how an approach into NPR effects might be achieved.

Blue Noise Dithering

A common thread between others’ works and this project is the inclusion of dithering to help stylize and mix colors within an image. This can be used even with binary images to represent a much wider palette of colors with just two colors such as black and white. [10]. A blog post by Bart Wronski titled *Progressive image stippling and greedy blue noise importance sampling* [13] demonstrates the effect this type of dithering can have on an image. In this post, Wronski shares a python program that recreates an image through a method he calls “progressive stippling”. [13].

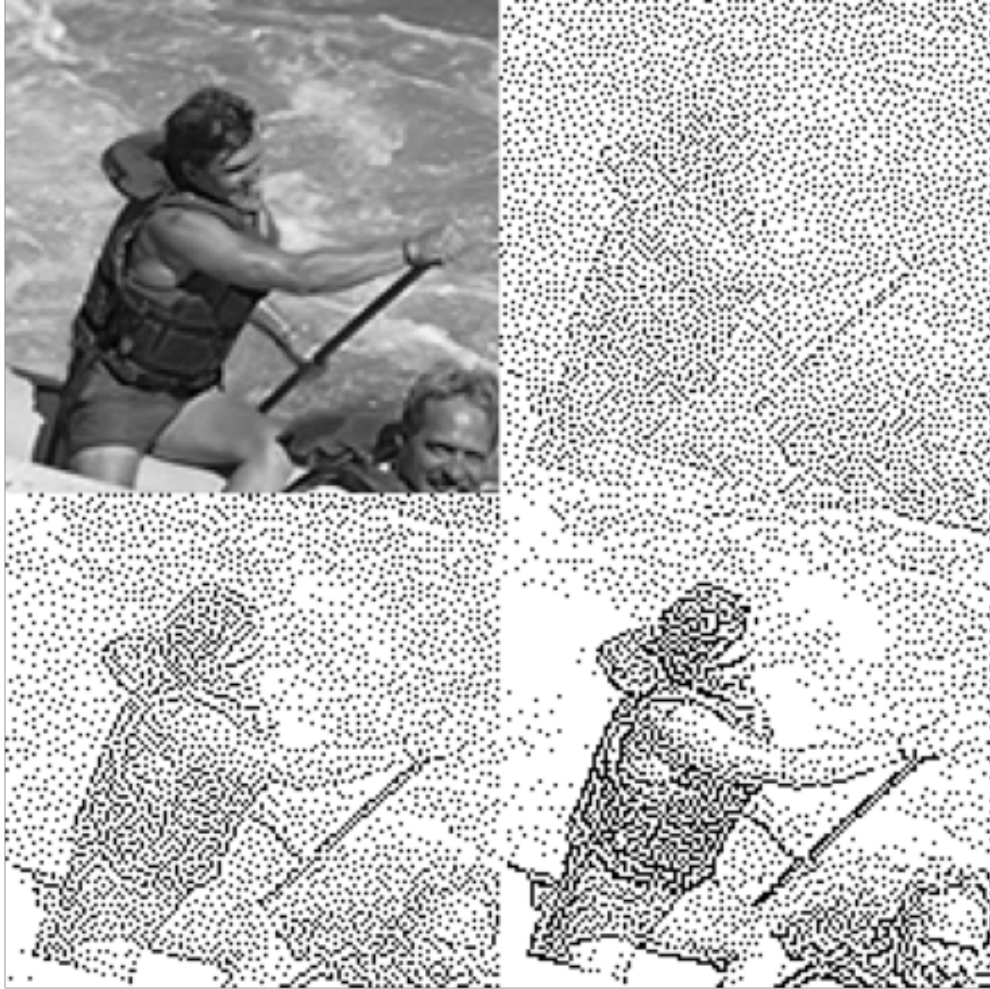


Figure 4: Progressive image stippling and greedy blue noise importance sampling example

This effect is very similar to an approach known as blue dithering dithering, and is more of an innovation on it than anything. [11]. This is done using a void-and-cluster algorithm that can generate blue noise which can then be used to dither, or stipple, an image. [12]. This project attempts to create a stippling NPR effect as part of the research, which also uses a blue noise texture generated by a void-and-cluster algorithm. Unlike Wronski's work, however, this project utilizes a static texture in the pursuit of real-

time feasibility. This means that while this project cannot directly generate its own unique noise texture during runtime, it can still benefit from the pseudo-random effects of blue noise.

Wronski’s work provided a good example of how blue noise dithering could be used to stylize an image and produce a visual artifact reminiscent of sketches. In doing so, it partially inspired this these in its pursuit of methods to control the shading and luminance of specific parts of an image without introducing addition colors. Although this theses decided to use ordered dithering as its primary method of approach, blue noise dithering was an important stepping stone to reach that conclusion.

Ordered Dithering

Another type of dithering that this project uses for the sake of color quantization – the reduction of the color palette in an image – is ordered dithering. Ordered dithering was first proposed by Bryce Brayer in 1973 in his paper *An optimum method for two-level rendition of continuous-tone pictures*. [1]. This paper proposed a solution in the form of dither patterns for two-level rendition of continuous-tone pictures. This solution can essentially be boiled down into a specific dithering pattern in the form of a matrix of varying sizes. For the sake of this project, a 4x4 matrix is used that is referred to as a Bayer 4x4 matrix, although there are bayer matrices of various sizes.

$$M = \frac{1}{4} \times \begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}$$

$$M = \frac{1}{16} \times \begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix}$$

By using these matrices, this work is able to take advantage of Bayer’s innovation and use his original work in interesting, albeit unintended ways. That being that using an ordered matrix for color quantization works exceedingly well, and allows non-photorealistic rendering algorithms to quantize color through various means and remove the subsequent color banding that may result. [15]. Furthermore, the use of a simple 4x4 matrix on the GPU keeps the cost of this dithering especially low in terms of performance.

This is because for each pixel on the GPU, that pixel only needs to access its surrounding neighbors to calculate its new color. This effect’s cost to performance is low, but does scale exponentially with the size of the matrix, which is why a 4x4 matrix is used when larger, more accurate matrices are available.

A bayer 4x4 matrix is used in the primary processing stack proposed by this thesis. This is done for the intended purpose of adding noise to the image in a recognizable, aesthetically pleasing way. Furthermore, ordered dithering has very little performance impact since it requires no complex computations or large textures. This makes it especially suitable to be used in a real-time environment and therefore ideal for this thesis implementation of non-photorealistic color quantization.

Color Spaces

A blog post written by Björn Ottosson in 2020 by the name of *A perceptual color space for image processing* [6] proposes the creation of a new color space for image processing. This color space, named Oklab, aims to be perceptually uniform without significant drawbacks to its use in image processing. Ottosson states, “A color in Oklab is represented with three coordinates, similar to how CIELAB works, but with better perceptual properties”. [6]. As the CIELAB color space is already suitable for image processing, this makes Oklab the ideal color space to be used with color quantization when converting images for human perception within the bounds of this project.

This color space is used in this project’s work in order to maintain a similar amount of luminosity – perceptual brightness – in an image before and after quantization, even when a palette’s available colors is modified. Comparing this to the RGB and sRGB color spaces, there are much less visual artifacts [14] present that can occur when switching out an image’s palette or quantizing it. This is because of the ability to freely calculate and change colors in an image, without needing to worry about specific instances of colors not quite matching up well with the perception of the human eye. This technique, when combined with dithering, provides a very seamless transition that enables this project to provide real-time color quantization at a large scale.

Without this proposed color space, this project would be forced to use a more traditional color space such as sRGB. While these color spaces would

still work for the purpose of color quantization, they would likely introduce some contradictions in regards to human color perception. For example, some darker shades of green may be quantized to a color containing a larger amount of blue or red due to the similar luminance, even when another shade of green is present that may be perceived as more suitable.

Where This Project Stands

This project combines the ideas, implementations, and theories of various works together in order to provide a new image processing algorithm for the purpose of non-photorealistic rendering in real-time applications. Although most of these works were done in isolation from one another, they each provide a key part of a greater solution that allows this project to change an application's visuals so that it may be perceptually more interesting in a given context. This is done through color quantization and palette swapping, dithering, edge detection, and even depth-of-field effects. Without the foundation each of these works provides in their respective fields, it would be much more challenging to solve many of the unique problems that arise in standard image processing, let alone non-photorealistic rendering.

Ethically, this project's research aims to combine a variety of lesser known techniques together in order to create a completed visual effect for the sake of a non-photorealistic rendering algorithm. It does not aim to simply copy and reproduce these effects, but instead to innovate on them and bring them into a new, real-time environment where they can be used for new, untested purposes. By doing this, this thesis can be used as a reference for real-time application developers to innovate in the field of NPR, and modify their products to use the discussed techniques.

Method of approach

Algorithm Overview

The goal of this project was to discover and develop a non-photorealistic rendering post-processing technique to be used in a real-time environment. This technique would take the form of an algorithm that could then be ported to any type of real-time application development software. Because of this, the algorithm and its components are not necessarily described in code but more by the ideas behind their implementations and the mathematics involved, although pseudo-code will be included when relevant. This ensures that the way the algorithm is described is not necessarily biased towards an existing language or the tools it provides, but rather on the logic and low-level ideas behind the development and implementation of it. That being the case, the images created for the sake of demonstration were developed using the Unity Game Engine and the C# and GLSL programming languages.

To find a suitable post-processing algorithm, various smaller, stylistic algorithms were researched in order to determine what would be possible in real time. Additionally, various non-technical art-styles were analyzed to see if their visual effects could be reproduced in a highly efficient environment. By looking at what smaller algorithms would work well in real time and forming an understanding for what the desired effect may look like, a larger, more comprehensive effect is able to be developed using a modularized format.

Since these effects are isolated and modularized, they are well suited to being included in a graphical pipeline where many separate effects can be layered onto each other in order to create a complete non-photorealistic rendering. Using this type of pipeline in conjunction with many smaller, individualized algorithms not only allows the greater algorithm to be easily modified and tested, but also enables smaller pieces to be ‘swapped out’ for similar techniques, ensuring that the overall effect of the work has a high de-

gree of developmental freedom at all stages of creation. An example of this would edge detection, which historically has many methods of implementation that often change depending on the greater context of the overall work, whether that be in terms of performance, visuals, or detail. By implementing the algorithm in this modularized setting, the different technical methods of implementation can be tried against each other in order to determine their suitability. For edge detection, this means comparing different algorithms such as Canny Edge Detection, Difference of Gaussians, and Depth-based edge detection. While they all achieve a similar technical goal, these algorithms each have their own technical strengths and weaknesses which is only fully understood once put in the full context of the overall processing effect.

Edge Detection

As mentioned, edge detection is an important step taken in the stylistic design of this project’s proposed algorithm. It is used primarily so that different objects and geometries within the scene can remain clearly outlined and visible even in an extremely noisy or mute visual setting. This is important as most real-time applications focus on interacting with, and in response to, a user. Because of this, the inclusion of edge detection as a means of outlining and highlighting the geometry present is a crucial step for these NPR algorithms so that the application can maintain a clear, easy to understand and read image for the sake of the user.

When considering edge detection in a pixel-based context, a strategy needs to be employed that can be used to accurately determine which pixels in an image represent edges. Normally, this would entail comparing a pixel to its neighbors by means of contrast and then determining some type of threshold that constitutes which pixels do and do not count as edge pixels. However, since we are dealing with real-time applications in 3D environments, we can leverage depth information in order to determine which pixels should count as edge pixels. This technique of using depth and normal information to determine where edges in geometry are is comprehensibly known as depth-based edge detection.

Depth-based edge detection is one of the most efficient, albeit simple, forms of edge detection because it requires a very small amount of passes – transformations run on an image through the use of shaders. The algorithm works by passing the depth and normal information from the scene – which

is often precomputed by an engine's camera – to the gpu where it can then be sampled by the shader programs to determine where edges are.

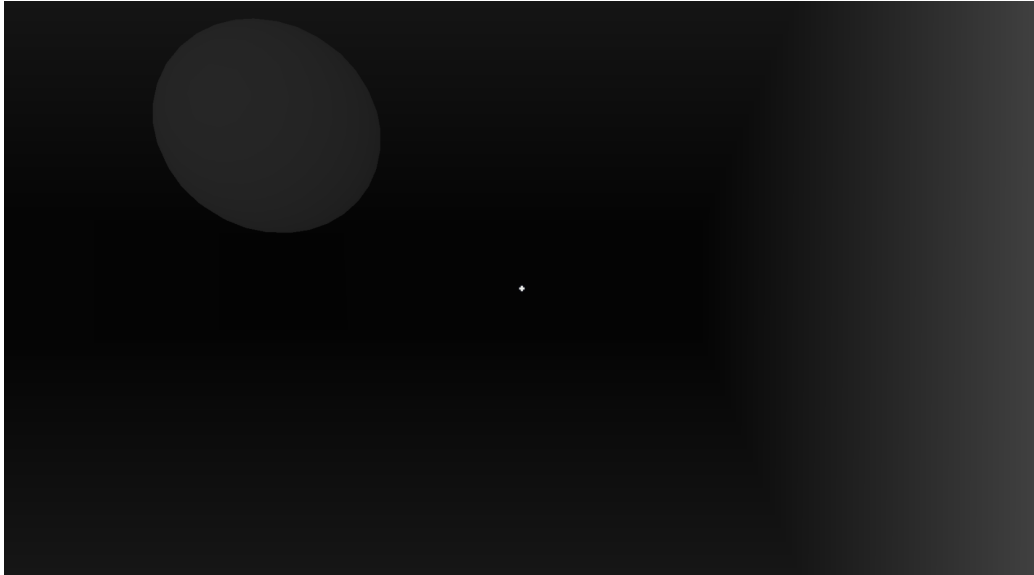


Figure 5: Depth Texture

This figure shows a visual representation of the depth texture, where we sample the depth of a pixel as its red, green, and blue color values. By passing this information to the gpu, we can compare each pixel to its neighbors and if there is a significant different in depth, mark the pixel as an edge pixel. In the case of this algorithm, marking a pixel simply means coloring it black, while unmarked pixel simply retain their original color. The following pseudo-code describes the logical steps of the algorithm in detail, as ran by the gpu to compute each pixel in an image.

```
FOR each neighbor pixel around center pixel
  IF neighborPixel.depth != centerPixel.depth
    pixelColor = black
    BREAK
  IF neighborPixel.normalsXYZ != centerPixel.normalsXYZ
    pixelColor = black
    BREAK
RETURN pixelColor
```

Breaking down the pseudo-code step-by-step, we start by initiating a loop to compare all neighboring pixels to the center pixel which we are currently determining the color for. This can be visualized as looping through a 3x3 grid where the center pixel is the one being modified. Each pixel in the grid is then compared to the center pixel by its depth value, and when there is a significant enough difference the center pixel is colored black and the loop breaks. Similarly, each of the normal values of the neighboring pixels are compared to the center, and when a difference is detected the pixel color is set to black and the loop breaks. If the loop is not broken by the time each neighboring pixel is checked against the center pixels, then the original color is returned. This process is independently and in parallel for each pixel being displayed on the screen. The following figure is an image of a scene before and after this shader pass with non-edges being displayed as white for clarity.

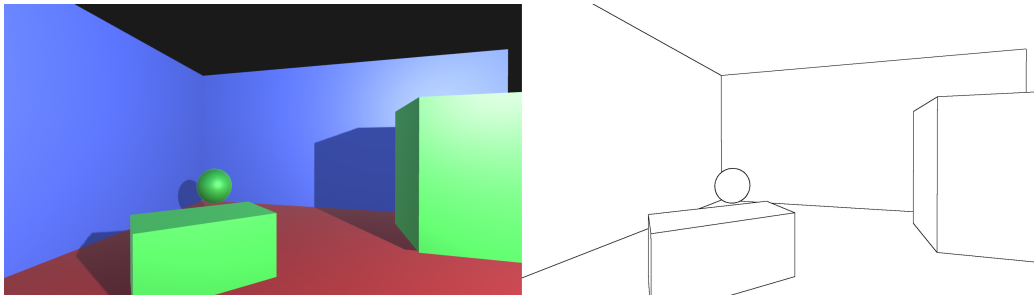


Figure 6: Depth Edge Detection

As shown by the figure, edges present in the scene are sharply and accurately defined in a way representing the scene's present geometry.

Color Quantization

Color quantization is the process of taking a large color palette and compressing it to a smaller quantity of colors. This is usually done with the goal of creating a more stylized, retro-based effect but has many applications in all kinds of projects. At a high level, the idea is that you take each pixel of a texture and change it to the color that it is closest to on a predetermined palette. In practice, we find that comparing colors to each other accurately is quite difficult. This is because each color, as computers see

them, are 3-dimensional or 4-dimensional vectors containing values for the red, green, blue and alpha values. This is commonly referred to as sRGB color formatting and is the most common way for computers to store color. Because of this, the simplest way to compare colors is by simply comparing the magnitude of each color vector to each other. This more or less works for comparing the intensity of two colors, but lacks the ability to differentiate the different color values, or hues, from one another.

To solve this problem, the shader needs to change the color space that is being used to compare colors. This would then allow the algorithm to then accurately map every color of the scene to a color from the palette while taking into account the precise, perceptual difference in hues. For my implementation of color quantization, I decided to use the OKlab color space. [6]. This color space maps colors in a way that allows them to be compared to one another by their perceptual difference, rather than their literal color values. By using this color space to compare and quantize colors, the shader can more accurately define and map colors based on how different they are visually from each other from the perspective of human vision. After converting a color to this space, the shader can then calculate multiple delta values which represent the distance between the pixel color being changed and each of the colors available in the palette. After this, the shader then changes the pixel's color to the one on palette with the lowest delta value. The quantization is finished once each pixel in the render has been changed to match a color on the predefined palette.

Similar to RGB, the OKlab color space is also represented through a 3-dimensional vector. The calculation used to transform the RGB pixel color values to the OKLab color space is

$$L = 0.2105 \cdot \sqrt[3]{l} + 0.7936 \cdot \sqrt[3]{m} - 0.0041 \cdot \sqrt[3]{s}$$

$$a = 1.9780 \cdot \sqrt[3]{l} - 2.4286 \cdot \sqrt[3]{m} + 0.4506 \cdot \sqrt[3]{s}$$

$$b = 0.0259 \cdot \sqrt[3]{l} + 0.7828 \cdot \sqrt[3]{m} - 0.8087 \cdot \sqrt[3]{s}$$

where

$$l = 0.4122 \cdot r + 0.5363 \cdot g + 0.0514 \cdot b$$

$$m = 0.2119 \cdot r + 0.6807 \cdot g + 0.1074 \cdot b$$

$$s = 0.0883 \cdot r + 0.2817 \cdot g + 0.6300 \cdot b$$

and r , g , & b represent the red, green and blue color channels of the pixel, respectively. It is worth noting that the transformation from RGB to OKlab assumes that linear RGB is being used, which in this case is implied. In the case an engine or software is being used that instead uses sRGB, the formula

$$r = \begin{cases} ((r + 0.055)/1.055)^{2.4} & \text{if } r > 0.04045 \\ \frac{r}{12.92} & \text{if } r \leq 0.04045 \end{cases}$$

$$g = \begin{cases} ((g + 0.055)/1.055)^{2.4} & \text{if } g > 0.04045 \\ \frac{g}{12.92} & \text{if } g \leq 0.04045 \end{cases}$$

$$b = \begin{cases} ((b + 0.055)/1.055)^{2.4} & \text{if } b > 0.04045 \\ \frac{b}{12.92} & \text{if } b \leq 0.04045 \end{cases}$$

can be used to convert to linear RGB. Finally, once we have two colors represented in the OKlab color space, we can calculate their difference with the formula

$$delta = \sqrt{(L_1 - L_2)^2 + (a_1 - a_2)^2 + (b_1 - b_2)^2}$$

which calculates the literal distance between colors in the OKlab color space. The following algorithm represents how each pixel in a palette might be quantized to a new, predefined set of colors.

```
originalColorOKlab = ConvertToOKlab(originalColorRGB)
FOR newColorRGB in newPalette
  newColorOKlab = ConvertToOKlab(newColorRGB)
  newDelta = calcDelta(originalColorOKlab, newColorOKlab)
  IF newDelta < lowestDelta
    closestColor = newColorRGB
RETURN closestColor
```

The algorithm above starts by converting the original pixel color from linear RGB to the OKlab color space using the previously described formulas.

It then starts to loop sequentially through each color within the predefined color palette, converting them to the OKlab color space at the beginning of each loop. Next, the algorithm calculates a delta value that represents the physical distance between the two colors within the OKlab color space. Using this delta value, it can then determine if the color currently being evaluated is the closest one so far in the palette. Once the loop terminates, the closest color will have been determined.

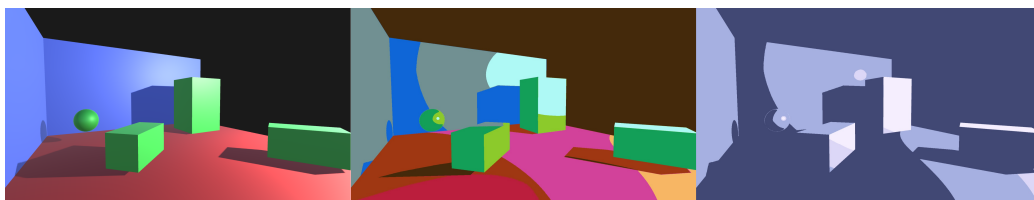


Figure 7: Color Quantization

The above figure demonstrates the color quantization effect in a scene with basic Unity geometry and coloring. The left image is the scene shown without the effect, the middle image is the scene quantized and rendered with 12 distinct colors, and the right image is the scene quantized and rendered with 4 similar colors. Even in cases such as the right image where there is not an obviously similar color available in the palette for the algorithm to quantize to, the image is able to be quantized and rendered in a recognizable manner. Of course, with just quantization much of the detail is being lost in the final render due to a common graphical issue known as ‘color-banding’. However, there are various additional algorithms that can be used to make up for this loss of detail without violating the strict rules of the quantization.

Dithering

The term dithering in graphics processing refers to the additional of noise to an image to help modify it in some way. In most cases, it is done with the purposes of mixing colors or imitating shaded regions in the image. This can be useful in environments that support a limited amount of colors or need to avoid large regions of flat color. Because of this, dithering has a large amount of potential in the development of NPR shader algorithms. The development

of this project experimented with two algorithms that can be used to dither the image in very different ways: blue noise dithering and ordered dithering.

Blue Noise Dithering

Blue noise dithering, as the name might suggest, achieves a dither-like effect through the application of blue noise. The reason blue noise is used is that it can create a visually interesting, pseudo-random pattern that works well in image processing applications. [11]. This blue noise is usually generated using a void-and-cluster algorithm [12], which creates a texture that can be sampled. This algorithm requires a lot of processing time however, and as such is unsuitable for real-time applications in most cases. Therefore, this project opts to instead use a single blue noise texture for its sampling in the described algorithm.

```
luminosity = 0.299 * pixelColorRed +  
    0.587 * pixelColorGreen +  
    0.114 * pixelColorBlue  
IF luminosity < noiseRed  
    color = black  
RETURN pixelColor
```

The above algorithm enables each pixel in a scene to be dithered according to its luminosity. This – through the use of blue noise – allows a stipple-like effect to be emulated. [2]. As such, this algorithm is quite powerful when attempting to add detail to images with a binary color palette such as black and white. It works by simply comparing the red channel values of the blue noise texture at the same pixel location as the render image to the luminosity of the pixel being modified. The luminosity is calculated using the formula

$$Luminosity = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

where R , G , & B represent the red, green, and blue color values of the pixel respectively. The figures below demonstrates the algorithm being run on an image quantized to just black and white, with the edges of the geometry marked for clarity.

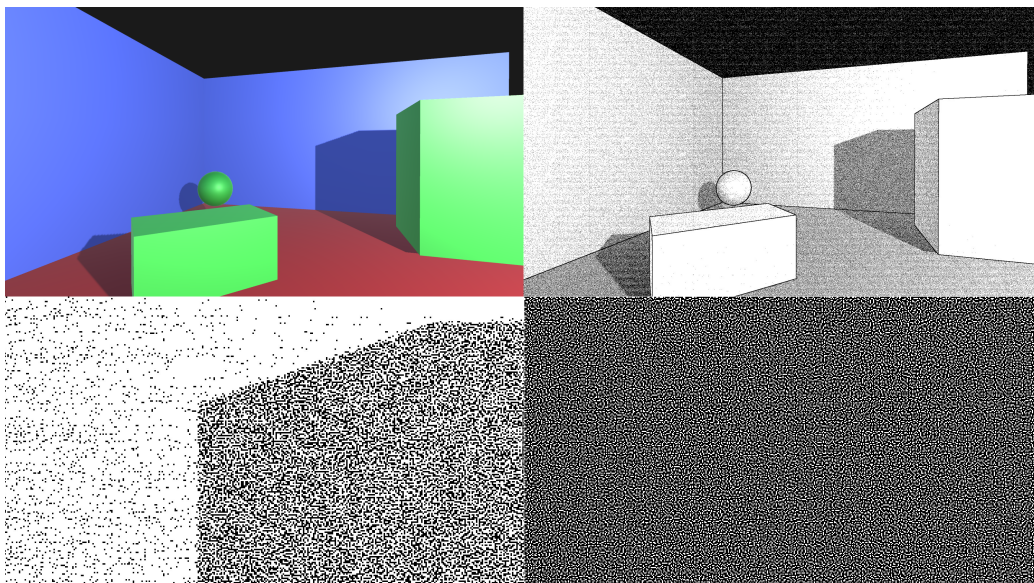


Figure 8: Top-left: No Post-processing, Top-right: Blue Noise Dithering, Bottom-left: Close Blue Noise Dithering, Bottom-Right: Blue Noise Texture

Ordered Dithering

As opposed to blue noise dithering, ordered dithering relies on a predefined matrix that can be used to calculate the effect. This has the effect of consistently blending the colors across the screen in a grid-like pattern. While this effect may not look as natural as blue noise dithering, it is stylistic and recognizable, allowing it to work well in the context of NPR algorithms. [10]. Additionally, the seamless nature of the effect allows the image to avoid the obvious tiling patterns that may plague the blue noise dithering effect at higher resolutions. This is because the matrix used blends into itself and does not create any unnatural edges. Finally, ordered dithering works exceptionally well in a real-time environment, as it does not rely on unique noise to create the effect. As such, no data needs to be transferred to the GPU from the CPU aside from the texture and color information, and no noise generation step is required.

As discussed previously, the matrix

$$M = \frac{1}{16} \times \begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix}$$

is referred to in this thesis as the bayer4x4 matrix. [1]. This matrix is simple in its construction and can easily be stored on the GPU for reference. While other bayer matrices do exist, the bayer4x4 matrix was chosen. This is because it is a small enough size as to not cause a significant increase in performance time, but detailed enough to enhance the dithering effect. The algorithm used for ordered dithering is outlined as follows:

```
matrixX = pixelCoordinateX % 4
matrixY = pixelCoordinateY % 4
color = color + colorSpread * bayer4[matrixX, matrixY] - .5
RETURN color
```

where colorSpread is calculated as

$$ColorSpread = \frac{3.0}{\log_2(c)} * s$$

and c represents the amount of colors that will be present in the final image and s represents the scalar by which to multiply the effect. In the case of this algorithm, a scalar of .2 is used. The algorithm starts by calculating which matrix value should be used. This is done by taking the pixel coordinate modulo 4 in both the x and y axes, as the matrix is of size 4 in both axes. Then the algorithm uses the computed color spread in addition to the matrix value to shift the color of the pixel. Finally, .5 is subtracted from the result in order to keep the average luminance of the image the same, and to avoid overly bright areas that could cause a loss in detail.

Processing Pipeline

Once all of these algorithms are researched and implemented into individual shader programs, a graphics processing pipeline can then be set up which will produce the non-photorealistic rendering effect. This is done in such a way where the first individual shader receives an image render of the scene,

runs a process on it to modify the output in some way, then returns it to the CPU to be passed on to the next shader file. Once every shader in the pipeline has modified the image sequentially, the final render is then returned and displayed on the screen. This setup allows different pipelines, or ‘stacks’, to be set up and pre-configured for the sake of testing and experimentation. The figure below represents the full processor stack for the non-photorealistic rendering algorithm, with a render taken at each shader pass.

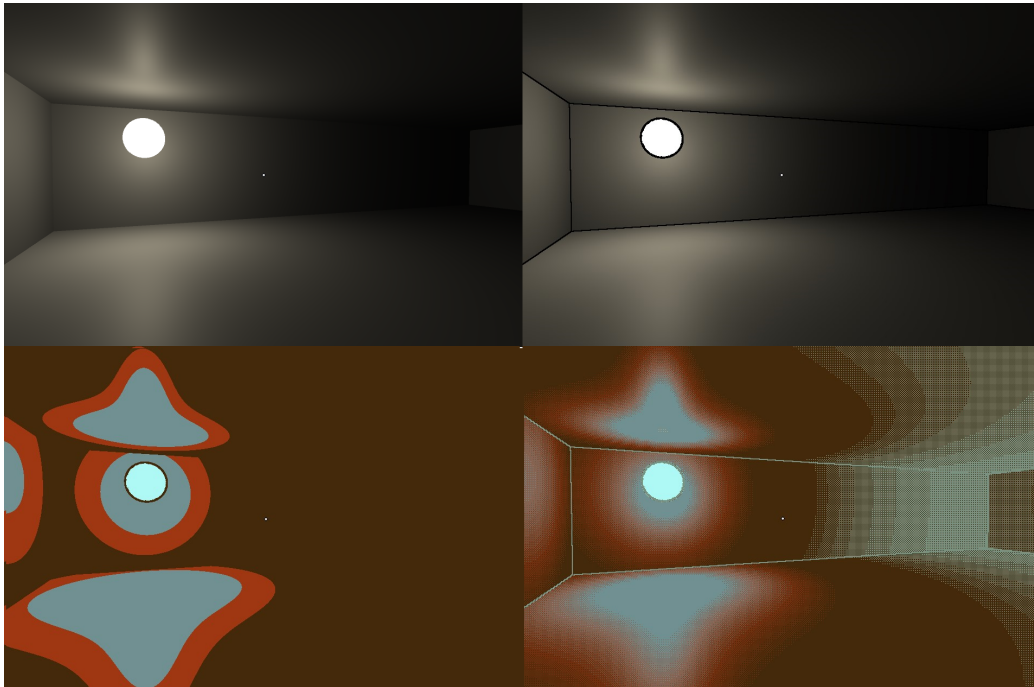


Figure 9: Top-Left: No Post-processing, Top-right: Edge Detection, Bottom-left: Color Quantization, Bottom-right: Color Quantization with Dither

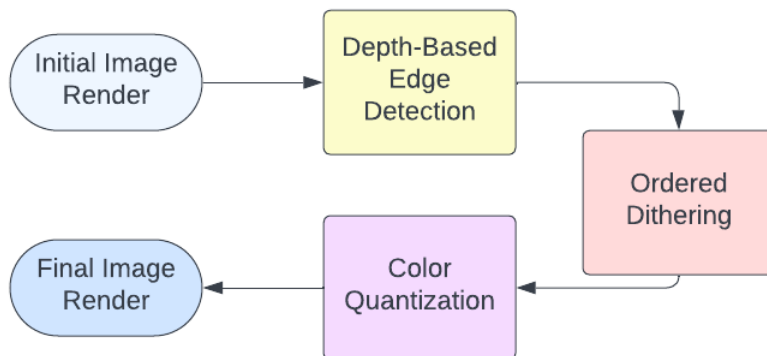


Figure 10: Graphics Pipeline Flowchart

The pipeline starts with a simple render of the scene (top-left) that contains an empty room and a light near the corner. The render is then transferred to the depth-based edge detection algorithm, which outlines the light and room geometry (top-right). Next, the render is sent to the dithering algorithm and then finally quantized to a predetermined color palette. The bottom left image represents the render if the dithering process is skipped while the bottom right image is the result when the image is dithered prior to being quantized. As shown by the figure, the dithering completely transforms the image from an abstract arrangement of colors barely able to represent the scene to a stylized rendering capable of representing the scene in high-fidelity detail while maintaining the limited color palette. Furthermore, the pipeline achieves a complete stylistic transformation that is both highly modifiable and flexible in a variety of environments. Below is a series of examples demonstrating how the algorithm might be used to change how the scene is rendered.

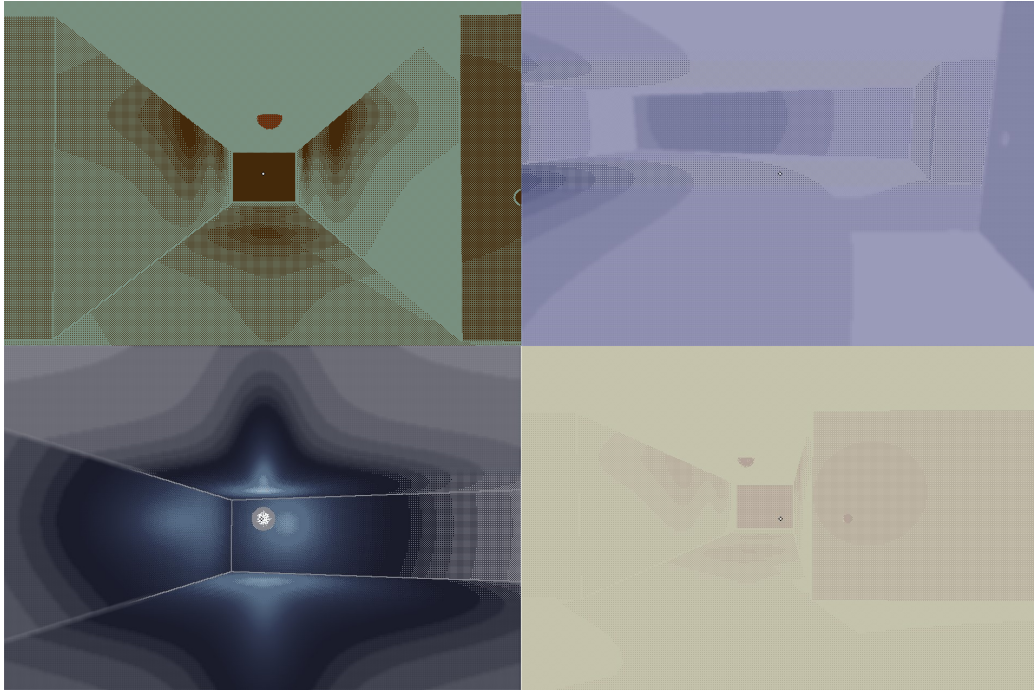


Figure 11: Top-Left: 12 Color Palette, Top-right: 4 Color Palette with DoF, Bottom-left: 16 Color Palette with DoF, Bottom-right: 4 Color Palette

Two of the images (top-right and bottom-left) use an additional shader pass that applies a strong depth-of-field effect to the image. This effect blurs parts of the image that the camera is not focused on using a Gaussian filter. When used with the dither aesthetic, it creates an odd effect where unfocused parts of the image feel blurred and smoother while the focused parts look as if they have more texture. These examples are included to show how the algorithm can be extended and implemented into existing graphics processing pipelines that use common, photorealistic filters like depth-of-field.

Running the Demo

The demonstration artifact source code can be found on github. The demo can be either opened as an independent Windows executable or imported into Unity.

How to Run

Windows Executable

1. Navigate to releases and follow the instructions for installing and running the latest release

Unity Project

1. Pull the github repository
2. Open the Artifact folder in Unity
3. Upgrade version or open with Unity 2021.3.4f1
4. Wait for shaders and scripts to compile
5. Press play

Modifying the Demo

- Create your own graphics processing stack in the Assets/ProcessorStacks folder through the Unity inspector
- Adjust shader parameters in the Assets/ImageProcessors folder through the Unity inspector
- Attach ProcessorStacks to the Player->Camera object

Experiments

Experimental Design

In order to facilitate the development and testing of the algorithms used in this project, the Unity Game Engine was utilized. Unity was chosen both because of its popularity in real-time application development and its ability to support graphics processing through its shader pipeline. Through Unity, each graphics processing algorithm was able to be implemented independently as its own shader file. A shader is a program written to be run on the GPU. Unity supports multiple frameworks and languages for developing GPU shaders. This project implements shaders through Unity’s shaderlab framework and the GLSL programming language. This implementation of independent algorithms allows a graphics processing stack to be formed as discussed previously as well as a way to modularize and test each of the algorithms in an isolated environment. Furthermore, parameters such as the color palette for the render to be quantized to can be changed and saved to a file so that tests can be done sequentially to measure performance.

When it comes to measuring the performance and proving the feasibility of the algorithms discussed in this thesis, certain metrics are used. The primary metric that this project concerns itself with is how long it takes to render and modify an image on the GPU before displaying it on a screen. There is no formal definition for how fast something needs to render to be considered ‘real time’ and the algorithms may be used in a variety of differing independent software. Because of this, the performance goals of these algorithms were to ensure that real-time is still achieved before and after the processing pipeline is used. This is quantized in the form of overall processing time per frame, usually measured in milliseconds. For the sake of human readability, however, a more intuitive format – frames per seconds (FPS) – is used. FPS simply measures how many overall frames the program is able

to render within a time frame of one second on average. This number is calculated by the formula

$$FPS = \frac{1}{\Delta t}$$

which divides one second by the time in milliseconds (ms) since the last frame was rendered. This number is calculated and recorded after every render. Doing this calculation every frame over an extended period of time yields results that give an accurate representation of how fast the render is completing in a real-time environment. By taking both the mean and median of these overall results, the different algorithms are able to be compared to one another as well as an identity result that represents the software running without any additional graphics processing steps. The experiments are able to be reproduced by running the included artifact and following the related instructions.

Evaluation

Cataloging the results yields the following tables:

Table 1: Performance Metrics for Each Algorithm Independently Over 30 Seconds Each. (Desktop)

Processing Pipeline	Mean FPS	Median FPS	% Relative to Identity (Mean)	Standard Deviation
Identity	1091	1154	100%	8.96%
Depth Edge Detection	921	937	84.4%	7.55%
Ordered Dithering	1083	1146	99.2%	9.04%
Color Quantize (12)	1022	1052	93.7%	7.95%

Table 2: Performance Metrics for the Fully Configured Processing Pipelines Over 30 Seconds Each. (Desktop)

Processing Pipeline	Mean FPS	Median FPS	% Relative to Identity (Mean)	Standard Deviation
Identity	1091	1154	100%	8.96%
Full Stack (4 Colors)	821	834	75.3%	6.20%
Full Stack (12 Colors)	812	825	74.4%	6.15%
Full Stack (16 Colors)	784	799	71.2%	6.24%

Table 3: Performance Metrics for Each Algorithm Independently Over 30 Seconds Each. (Laptop)

Processing Pipeline	Mean FPS	Median FPS	% Relative to Identity (Mean)	Standard Deviation
Identity	488	340	100%	46.64%
Depth Edge Detection	480	328	98.4%	49.97%
Ordered Dithering	486	406	99.6%	46.83%
Color Quantize (12)	485	422	99.4%	48.01%

Table 4: Performance Metrics for the Fully Configured Processing Pipelines Over 30 Seconds Each. (Laptop)

Processing Pipeline	Mean FPS	Median FPS	% Relative to Identity (Mean)	Standard Deviation
Identity	488	340	100%	46.64%
Full Stack (4 Colors)	475	388	97.3%	53.15%

Processing Pipeline	Mean FPS	Median FPS	% Relative to Identity (Mean)	Standard Deviation
Full Stack (12 Colors)	473	372	96.9%	53.47%
Full Stack (16 Colors)	470	377	96.3%	53.55%

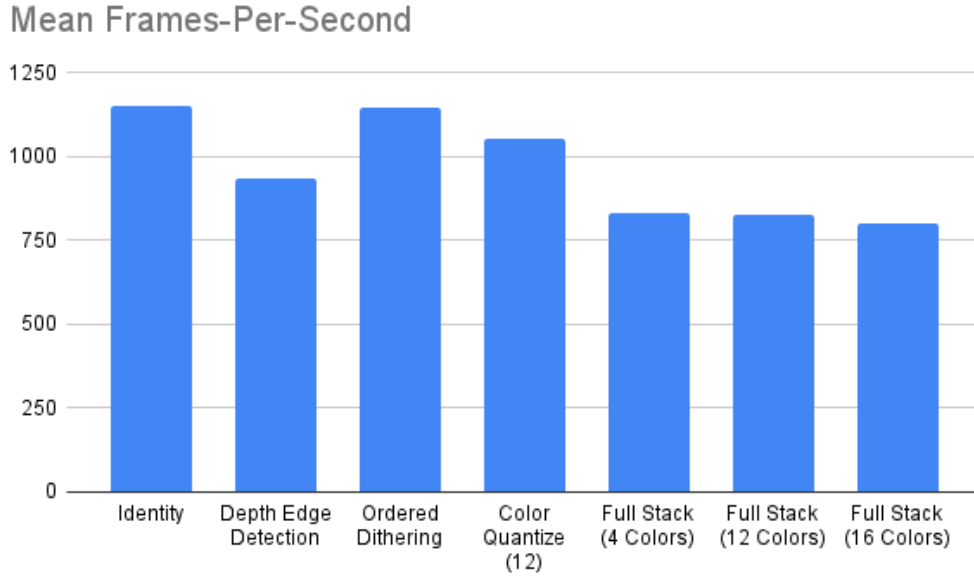


Figure 12: Mean Frames-Per-Second

Table 1 represents the processing time of each render when only one core step in the algorithm is being used at a time while table 2 represent the processing time of the full stack with a different amount of colors being represented in each processing stack. Tables 3 and 4 represent different values on a less powerful, more volatile device. An ‘identity’ stack is also used as a control case in order to compare and evaluate the performance of the other algorithms. The identity processing stack does not modify the image in any way through graphics processing programs or algorithms, but still runs the surrounding framework to ensure that consistent results are recorded.

The four values that this thesis considers important to measure performance with in each processing stack is the mean frames-per-second and its standard deviation, the median frames-per-second, and percent deviation from the identity on average. The mean value enables the experiment to determine if, on average, real-time performance is being met. The standard deviation of the mean is also calculated in order to show how consistently close the results are to the mean in general. This is done by using the general standard deviation formula

$$\sigma = \sqrt{\frac{\sum (x_i - u)^2}{N}}$$

The median is calculated in addition to the standard deviation to ensure that there is not a significant skew in data, as it should be relatively close to the mean, if not slightly greater due to the immediate lower performance that occurs when transitioning processing pipelines. Finally, the percent deviation from the identity is perhaps the most useful value to consider as it represents the expected performance impact when using the algorithms in a variety of settings. Furthermore, it is the single value that should be expected to be consistent across multiple different devices and hardwares, which is an important consideration when considering deployment in real-time application development. This value is simply a comparison of the identity and current stack in terms of mean performance by dividing the identity performance by the stack performance to determine the relative decrease. All experiments represented in table 1 and table 2 have been run using a device with the following specs:

Desktop-QCNNQRE
Windows 10 Pro 64-Bit
NVIDIA GeForce RTX 2080 SUPER
Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz
16.0 GB DDR4 3200 MHz Memory

Tables 3 and 4 are run with a portable device with the following specs:

LAPTOP-GE0B4FCS
Windows 10 Pro 64-Bit
NVIDIA GeForce GTX 1660 Ti with Max-Q Design
Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz
16.0 GB DDR4 2600 MHz Memory

Analysis

Looking at the performance cost of the individual graphics rendering algorithms in table 1 we can determine exactly which steps are taking the most processing time to complete. Depth edge detections especially seems to be an expensive processing step, as it lowers the frames-per-second in the application from a mean of 1091 FPS to 921 FPS. This is more than a 15% decrease in rendering time, which is quite a significant cost to performance. This is likely due to the required additional depth and normal textures that Unity needs to first generate and then sample. However, while the cost in performance for this method of edge detection is quite steep, it is still highly accurate since the algorithm uses the physical geometry of the scene to generate the data for where the edges should be located.

Unlike the depth edge detection step, the ordered dithering step has almost no impact to the application’s performance. On average, the algorithm only lowered the performance by 8 frames, going from a mean 1091 FPS to a mean of 1083 FPS. This is only a .8% decrease in performance, which is well within the standard deviation. Since ordered dithering uses a predefined matrix and avoids needing to do any complicated calculations, it ends up being very cheap to run in parallel on the GPU, leading to no significant loss in performance by itself.

The color quantization step *does* have a more significant performance impact than the ordered dithering step, although it is still much less substantial than the cost to performance present in the depth edge detection algorithm. It lowers the mean fps from 1091 to 1022, which is only a 6.3% decrease in performance. While not insignificant, this cost of performance for a graphics rendering step that has such a transformative effect on the final image is relatively cheap. This experiment was run assuming a palette of 12 colors, which is more than a super limited palette would be but also less than an expansive color palette, and so is suitable as a moderate choice. As table 2 shows, while the amount of colors present in the palette does increase the cost to performance, it is only to a lesser extent.

The data in table 2 represents the cost to performance when running the entire processing pipeline with different color variables to serve as the quantization palette. Across the 3 difference pipeline tests, a performance cost between 20% and 30% is measured. These results indicate a very substantial cost to performance, which is to be expected when rendering an image through multiple shader passes in every frame. Since the results of the full

pipeline on the image in real-time are completely transformative, this cost could be easily justified, but is still large enough to warrant due consideration before being implemented in any project. The performance cost of the algorithm is largely consistent, but does tend to increase as the size of the color palette increases. Analyzing the results of the experiments, this increase is able to be determined as linear to an extent, and so is worth considering if a suitably large color palette is used.

Tables 3 and 4 also show a loss in performance when running the individual processing stacks, although the resulting increase in run time per frame is much less drastic. This is due to the fact that the data in these tables was recorded using a less powerful device that likely bottlenecked the performance of the program on the CPU side. Because of this, the GPU was generally underloaded relative to the CPU and so the addition of extra graphics processing steps made little difference in the total processing time of each render. Also worth noting is the loss of consistency in performance, which is shown through the significantly lower median value for each stack, as well as a much higher standard deviation of $\sim 50\%$. This is likely also a result of various bottlenecks introduced through the use of a less powerful system. Although these results are less consistent overall when compared to the more powerful device, they still show that the various algorithms involved are not prohibiting the application from achieving real-time non-photorealistic rendering.

Threats to Validity

While the results of the experiments were very detailed and informative, one aspect of uncertainty, as mentioned previously, is the dependency on external hardware to run and analyze the graphics processing algorithms. This dependency tends to result in many different types of hardware and software being used in combination. As such, the results between individual devices will be largely very different when looked at literally. However, some of the results will still be somewhat consistent, such as the standard deviation and the relative performance when compared to the identity metrics. This greatly helps normalize the problem and allows the algorithms to be tested reasonably well on a single device, but does not solve it completely. However, even these values can vary greatly across different devices, as shown in the data generated using the less powerful, portable device. The only true way to

ensure consistent performance across most operating systems and hardware would be large scale testing, a feat that is vastly outside the scope of this thesis. Still, while the testing is only able to be run in a limited capacity relative to the different types and combinations of hardware available, it is still largely indicative of the performance of the individual algorithms, as well as the full graphics processing pipeline.

Conclusion

Summary of Results

As proposed by this thesis, an algorithm has been developed that executes as a series of post-processing steps towards the end of a graphical processing pipeline. Through this algorithm, non-photorealistic rendering is achieved in a real-time environment. The algorithm stylizes the video output through techniques such as depth-based edge detection, ordered dithering, and color quantization. In the development of this algorithm, various smaller, independent algorithms were individually researched and developed. This was so that the thesis could form these algorithms into an image processing pipeline, which could easily be modified and tested for the sake of optimization and ease of use. Among these algorithms, many additional effects were researched that did not make it into the finalized non-photorealistic rendering (NPR) effect, such as blue-noise dithering, depth-of-field, and Gaussian blur. While these effects were not implemented into the finalized post-processing stack, they played an important part in the research and development of the completed pipeline, as they gave an important perspective on how alternative rendering effects would perform both in terms of visual style and computation run-time.

The graphics pipeline ultimately detailed in this thesis enables real-time application developers to be able to algorithmically stylize and design their real-time applications. This ability to automatically alter the final image being displayed in real-time allows developers to save resources in asset development as they can now completely alter the visual style of an application without needing to recreate or alter existing assets. Furthermore, this method of art-design and visual stylization allows for a greater flexibility within the software being developed, as the individual properties of the visuals can be easily changed, even during runtime. This could have many

implications depending on the application, and could even be integrated into the user sphere of the application, allowing the end user to customize the look of the application in some way. Additionally, the description and implementation of these algorithms as individual pieces that are then combined via a graphics processing pipeline allows for a great amount of flexibility for developers to modify, customize and change the algorithms being used. These algorithms are able to easily be integrated with and combined into existing post-processing stacks so that they can work with and alter various other image processing effects such as depth-of-field and anti-aliasing. This is especially true when considering the unique properties of the described effect, as color quantization combined with ordered dithering is able to create very interesting unique and stylized visuals when combined with other graphics processing effects, depending on the precise implementation.

While the implementation of the proposed algorithms certainly had a significant performance impact in some instances, the tested software was easily able to maintain its real-time performance. Specifically, the experiments showed that the full algorithm could have a significant impact to performance when trying to render at an exceptionally high speed. However, as the processing power of the device falls, the relative performance impact is much less notable. The performance across all algorithms and both devices never dipped below real-time performance, the lowest point being the full processing stack with 16 colors with a mean of 470 frames-per-second and a median of 377 frames-per-second on the portable device. With a standard deviation of $\sim 53\%$, the lowest FPS that could be statistically expected, although rarely, would be 220 FPS, a number that is still significantly higher than the consumer standard performance in video games of 60 FPS.

Future Work

In the future, full-sized application development with this algorithm designed as the core focus and art-style would allow for more detailed research and experimentation, as well as a greater precedent of use of non-photorealistic rendering algorithms being used in real time. Currently, NPR effects are rarely used in such an all-encompassing capacity, and so it would be important to provide a direct, more complete example of how these effects could be used that would also double as a proof of concept. In a similar vein of thought, this graphics processing pipeline could also be implemented into ex-

isting real-time applications that allow – either directly or through 3rd party extension – the additional modification of the final output render through shaders. This would allow the program to be tested in a real-world environment without the expensive requirements present in real-time application development.

Alternate implementations of the algorithms involved that aim to achieve a similar graphical effect could also be developed. These might be developed for the purpose of additional flexibility, optimization, or a subtle change in style. One example of an alternate implementation would be to change how edge-detection is performed. Currently, edge detection is done with a depth-based approach, taking advantage of the depth and normal information that is present in most real-time application development environments. Rather than using this depth based approach, an algorithm such as Canny edge detection or the difference of Gaussians could be used. In doing so, the precise effect of the edge detection would be different; flat surface edges such as shadows may be outlined as part of the algorithm and the performance impact would be significantly different. Whether the alternative algorithm would be an improvement would largely be up to the developer, as various styles might be more suitable for different applications, regardless of the differing performance impacts.

Finally, one additional, yet optional feature that could be developed as part of this NPR algorithm would be the addition of high-dynamic range (HDR) color support. For the sake of accessibility, the proposed implementation of the algorithm only supports low-dynamic range. HDR color support would greatly improve the varieties of the colors being shown when displayed on a suitable device. Since this algorithm deals directly with color, implementing this feature in theory would not take too much additional shader development, and could have some drastic visual improvements in specific circumstances.

Future Ethical Implications and Recommendations

As the art and visual direction of applications is able to be more and more automated, one ethical concern that arises is the potential displacement of artists and digital asset creators in the field of software development. The

research proposed in this project may in some ways contribute to this displacement, as it may allow – in part – the replacement of asset creators by means of graphics processing algorithms. That being said, these algorithms and graphics processing pipelines are intended to augment, not fully replace, the traditional development of sprites and 3D models that accompanies real-time application development. While it would allow for developers to achieve more visually without artists, it still lacks the subtle requirements of object creation that could only be met at competence by a human hand. In fact, professionals involved in the asset creation process should be made intimately familiar with the graphics processing pipeline, so that they may focus their work on the interactions between the initial and final render of a given scene. In doing so, they would tailor their work to be used by the pipeline, or could even leave some details of the work such as color and texture to the pipeline to focus on the creation of additional sprites or models. In short, the NPR algorithm is not meant to replace artists, but automate some of the visual art development process, so that other aspects that cannot be done automatically have the chance to be more streamlined.

Another concern introduced by these post-processing implementations is the question of accessibility. By integrated these algorithms into an otherwise pre-existing project, an additional computational burden may be placed upon the user’s personal device. In some cases, it is conceivable that this additional burden may lead to the exclusion of some users that are running the software on a less capable device. Currently, the only solution to these concerns would be to include accessibility options within the software that change, limit, or deactivate the additional graphics processing steps. In including these options, developers using this approach to artistic rendering would be able to safely develop with it without needing to be concerned about limiting their potential user base. This being said, the performance impacts introduced by these techniques, especially on less powerful devices, are very slight, and should rarely – if ever – turn a user away on their own.

Conclusions

The proposed algorithms and subsequent graphics processing pipeline allow an application to stylize in real-time the final image render each frame through the use of techniques such as edge-detection, ordered dithering, and color quantization using the OKLab color space. In doing so, the application

allows developers to create a unique, independent art-style that is partially free from the reliance of additional asset creation. This in turn enables a greater flexibility by developers to test, develop, and implement various visual styles in accordance with their desired art direction. Furthermore, the existence of the proposed NPR effect as individual graphics processing algorithms present in a stack enables them to be integrated into various existing environments and post-processing pipelines. Finally, while there is a somewhat significant performance impact involved in the implementation of these algorithms, they do not threaten the goal of real-time rendering present in the applications they propose to augment.

References

- [1] Bayer. 1973. An optimum method for two-level rendition of continuous-tone pictures.
- [2] Jose Esteve. 2011. Stippling and blue noise. <https://www.joesfer.com/?p=108>.
- [3] Hasu. 2018. Fundamentals of shaders with modern game engines. Lappeenranta University of Technology.
- [4] Lee Kim. 2019. Layered non-photorealistic rendering with anisotropic depth-of-field filtering. *Springer Nature* 79, (2019), 1291–1309.
- [5] Milán Magdics, Catherine Sauvaget, Rubén J. García, and Mateu Sbert. 2013. Post-processing NPR effects for video games. In *Proceedings of the 12th ACM SIGGRAPH international conference on virtual-reality continuum and its applications in industry (VRCAI '13)*, 2013. Association for Computing Machinery, New York, NY, USA, 147–156. <https://doi.org/10.1145/2534329.2534348>
- [6] Ottosson. 2020. A perceptual color space for image processing. *bot-tosson.github.io*.
- [7] Salviado. 2023. Unleashing the power of shaders for generative art: An inside look at the creation of 'shoals'. <https://www.fxhash.xyz/article/unleashing-the-power-of-shaders-for-generative-art>.
- [8] Saqib. 2023. Incorporating cutting-edge graphics technologies in AAA game development using c++. *www.mycplus.com* (2023).
- [9] Schofield. 1993. Non-photorealistic rendering: A critical examination and proposed system. (1993).
- [10] Surma. 2021. Ditherpunk. <https://surma.dev/things/ditherpunk/>.
- [11] Wedekind. 2022. Blue noise dithering. <https://www.wedesoft.de/software/2022/09/21/blue-noise-dithering/>.

- [12] Wronski. 2021. Superfast void-and-cluster blue noise in python. *bartwronski.com*.
- [13] Wronski. 2022. Progressive image stippling and greedy blue noise importance sampling. *bartwronski.com*.
- [14] Xiang. 1997. Color image quantization by minimizing the maximum intercluster distance. *ACM Transactions on Graphics*.
- [15] Yliluoma. 2021. Joel yliluoma’s arbitrary-palette positional dithering algorithm. <https://bisquit.iki.fi/story/howto/dither/jy/>.