A comparison between techniques for color grading in games

Mattias Oldenborg
A comparison between techniques for color grading in games

Submitted by Mattias Oldenburg to Högskolan i Skövde, as a dissertation towards the degree of B.Sc. (Kandidatexamen) by examination and dissertation in the Department of Communication and Information.

2006-06-07

I certify that all material in this dissertation which is not my own work has been clearly identified and that no material is included for which a degree has already been conferred on me.

Signed: ____________________________________________
A comparison between techniques for color grading in games

Mattias Oldenborg

Abstract

Color has been significant in visual arts for as long as the art-forms have existed. Still images and movies have long used colors and color grading effects to affect the viewer and characterize the work. In recent years attempts have been made to bring these techniques of stylizing also to interactive games. This dissertation aims to compare two different approaches of performing real-time color grading for games. Focus is put on examining the two ways from a number of different perspectives and from there draw conclusions on advantages and disadvantages of the approaches. The results show no unanimously superior approach but rather aim to break down the results in categories and attempt to explain the benefits and drawbacks in using either one of them, aiding the decision for anyone inclined to implement color grading effects in games.

Keywords: Color grading, color correction, real-time, interactive games, pixel shader, lookup table
1 Introduction

Visual arts have long made use of color to evoke different emotions. Basically color grading or color correction concern the altering of color balance in post-production. This used to be achieved by applying a chemical treatment to the film, but with new technology, it is possible to achieve these effects digitally without permanently altering the original footage. Actually the first experiments with color alteration were done even before color film came around. *Kinemacolor* was the first successful color motion picture process, used commercially from 1908 to 1916. It was a two-color additive process, photographing and projecting a black and white film behind alternating red and green filters (*Kinemacolor*, 2006).

Color correction effects can be used to set a tone, reinforce a mood, or define a style. Obvious examples of color grading in feature films can be seen in *O Brother, Where Art Thou?* (Coen et. al., 2000), *The Matrix* (Wachowski et. al., 1999) and *Sin City* (Rodriguez et. al., 2005), just to name a few. Even though digital intermediate processes haven’t been used for more than two to three decades photo-chemical processes have been used as far back as the 1920’s. In the early days of digital intermediates, processing was usually only done on individual shots or sequences. The reason for this was the slow speed of the scanners along with large file-sizes compared to the available processing power. The first feature film to be digitally graded in its entirety was the movie *O Brother, Where Art Thou?*, made in 2000. However, the trend caught on quick and this year it is expected by Digital Intermediate (2006) that 90% of all Hollywood films are to go through a digital process. An important reason for the success of digital color design over photochemical is that digital grading allows the mixing of warm and cold colors which according to Jeunet, J-P. (2001) is not possible with photochemical color timing\(^1\), this of course give more freedom to movie creators. Another central reason is that a digital process only has to be carried out once whereas the older practice usually means photo-chemically color timing the film version and then brushing up the movie digitally for the DVD version. The photochemical process won’t be discussed further in this paper since this way is not applicable for non-digital color grading.

Since the introduction of programmable consumer class graphics hardware some efforts have been made to use the GPU to perform these effects. An early example can be found in *Advanced Image Processing with DirectX® 9 Pixel Shaders* by Mitchell et. al. (2003), where some techniques are presented that is part of the implementation for this project. Today, a variety of video compositing tools support effects being carried out by the graphic card instead of the main processor. For games however, these effects have not yet made a big breakthrough. Simpler and more limited effects have been available in a number of games, e.g. a game as old as *Doom* (id Software, Inc., 1993) had a “hurt” filter that blended the screen with red as an indication of taking damage. Although some examples of use apparently can be found more complex color transformations are not being utilized, especially not in a substantial amount. A reason for this may be performance concerns, the graphic cards have the power to support real-time color correction in for example a video compositing tools. Although, the performance hit may

\(^1\) Color timing is equivalent to color grading but the term is usually only used to describe the photochemical process, not the digital.
become too big in a game where a lot of other procedures have to be carried out by the GPU. The processing power of video cards is however one of the most rapidly increasing areas of modern computers. The time might therefore have come when bringing color grading effects into a game engine environment is feasible. Another reason that color correction tools are not directly available in game engines is that many of the effects can be achieved through various tricks in games. Since a developer have control of all properties in a game, as opposed to in photography and movies, changing the source artwork can well achieve a comparable effect. A motivation opposing this way of working is the simple matter of time, implementing color correction directly in a game engine will lower the turnaround time for changing the artwork in game development.

There could be many advantages to using color grading techniques in game development. Color can be used to draw attention to a specific part of the level or reinforce a feeling, such as suspense or fear. It is also very useful for the reuse of content as color correction tools can enable very quick alterations of the look to a scene. This will make it possible to change the style late in the development process without the costly modification of assets. Color correction for games will also give more control to the art director in matching the look within and between scenes.

Two examples of color grading features can be found in commercial game engines today. The engine Unity game editor, (2006) implements some color grading effects such as color-correction, grayscale and sepia filters procedurally as a post-process through its shading engine. Another approach can be found in the Valve Source Engine, (2006) where according to Gasior, 2005 color transformations are not realized through the use of pixel shader programs. The Source engine is said instead to rely on 3d texture lookups to implement these features. One game that can be noted down to use color grading is Red Dead Revolver (Rockstar Games, Inc, 2004). The game use these effects in its aims to imitate the look of spaghetti western movies, unfortunately no specific information seems to exist on how the techniques are implemented and made use of in-game.

The undertaking of this project is to implement color grading techniques by both an arithmetically, i.e. procedurally driven method and by a lookup table-powered approach and then to compare these two rather different paradigms on a number of selected aspects.

1.1 Target reader

The aim of this dissertation is to look at an issue in the area of real-time graphics, so it will mainly be of interest for programmers that work with programming graphics for games. It may also be of a more general interest to anyone involved in computer graphics. The work may be of some interest for use in applications that doesn’t have real-time restrictions, in this case for the purpose of making the application more efficient. The main focus will however be on implementing and evaluating algorithms that are useful in a game development environment. As one of the reasons for implementing these techniques in a game engine is to accelerate the development process it may be of some interest from a management perspective even though no concrete evaluation will be made in this area. The work will also touch on areas related to artistic side of game development but no real discussion will be made on why to use certain effects for certain purposes. It could have some relevance to an artist but the essence of the work will be on the technical aspect color transformations. More specifically some knowledge is required
in the RGB color space, pixel shaders (fragment programs), an experience in image editing applications is a benefit but not a necessity as essential concepts of this area will be explained.
2 Background

This chapter will present concepts that are central to the field of color grading and color correction. A range of various ways to adjust color are available and they usually differ slightly between different applications. Some basic color operations and some more advanced tools used for color correction will be presented. Further will the area of hardware image processing be presented with a rundown on the two different approaches that is at the core of this dissertation.

2.1 Color operations

To begin with a default input-output graph will be introduced, i.e. an input-output curve that do not influence an image in any way.

![Default input-output mapping](image)

**Figure 1: Default input-output mapping**

*Brightness* is defined by the CIE (*Commission On Illumination*) as the attribute of a visual sensation according to which an area appears to emit more or less light (Poynton, C. 1997). *Brightness* perception is a very complex topic and therefore this work will stay away from any deeper involvement into that discussion, any reader who wish to dive into the area is recommended to read the aforementioned FAQ by Poynton. A *brightness* operation in this dissertation refers simply to the mathematical functions of adding to, subtracting from, multiplying and dividing pixel values to change the intensity of the pixel, this corresponds to the *brightness* concept used by Brinkman (1999).

![Brightness operator](image)

**Figure 2: Brightness operator**
The *Gamma* operator is one of the more popular color correction operators according to Brinkman (1999), it’s strength lays in that it only effects values between 0 and 1 i.e. nonblack and nonwhite pixels. This makes the operation primarily affect midrange values meaning that blacks don’t become washed out and bright areas are not clipped to pure white which can be the case with linear operations such as add and multiply.

![Figure 3: Gamma operator](image)

The *Contrast* operator is a tool for changing the brightness relationship between the upper and lower color ranges of an image (Brinkman, 1999). There are no universal standard for how the *contrast* operator should be implemented. A very rudimentary way to heighten the *contrast* would be subtract a constant value and then multiply by another constant or to lower the *contrast* adding a constant value and then divide by another constant. A better way would be to apply gammalike curves to the upper and lower ranges, giving the picture a smoother look, particularly in the low and high ranges of the *brightness* spectrum.

![Figure 4: Contrast operator](image)

The *Invert* function causes all colors in the active layer to be inverted in brightness, as though the image were converted into a negative. Dark areas turn to bright and conversely; hues are replaced by their complement (The GIMP Documentation Team, 2006). Mathematically this is as simple as subtracting the RGB value of a pixel from an all white pixel, i.e. RGB(255,255,255).
A threshold function can be described as a black or white operation, based on a selected intensity value it transforms all pixels above the intensity threshold to absolute white, i.e. RGB(255,255,255) and all pixels below the intensity threshold to absolute black, i.e. RGB(0,0,0).

Another function that fits into the category of basic operations is the grayscale conversion function. This operation has no figure to illustrate its function since it doesn’t quite work like the above described effects. Normally the grayscale output for each color channel is achieved by mixing the input of all three color channels. The easiest way to do this would be to simply mix the channels equally, taking all color intensity, i.e. saturation out of the image (see Section 2.1.1 for more info on saturation). The problem with this approach is that the human eye does not experience the same intensity for different colors. Therefore blending by weights is generally accepted as the best way to convert an image to grayscale, and it can be set to match the sensitivity of human eyesight (Bjorke, 2004).

2.1.1 Some notes on the Hue/Saturation color space

There are multiple definitions of hue, and some are vaguer than others. The most precise definition is probably that hue is the attribute of a color by which we perceive it’s dominant wavelength. A more useful and equally valid definition of hue is that it’s the attribute of a color that gives it its basic name, such as red, yellow, purple, orange, and so on (Fraser et. al., 2003).
Saturation refers to the purity of a color, or how far it is from neutral gray. If hue is the perceived dominant wavelength, saturation is the extent to which that dominant wavelength seems contaminated by other wavelengths (Fraser et. al., 2003). Red can be said to be a hue but pink can not, it is more accurately described as a desaturated red.

The value component can be thought of as the brightness of the color, although strictly speaking it is defined as the maximum of red, green, or blue values.

**Figure 7:** The figure depicts the Hue-Saturation color space as an upside-down cone.

The HSB (Hue-Saturation-Brightness) and HLS (Hue-Lightness-Saturation) color spaces were developed in an age when users had to specify colors numerically. The usual formulations of HSB and HLS are flawed with respect to the properties of color vision (Poynton, 1997). In this dissertation the HSB (also called HSV) color space will be used for effects that require this alternative color representation. The reason behind converting between HSV and RGB, as opposed to storing color in HSV permanently is that there are some problems with these color spaces. For example HSB and HLS make no reference to the linearity or nonlinearity of the underlying RGB, and make no reference to the lightness perception of human vision. The usual formulation of HSB and HLS compute so-called “lightness” or “brightness” as (R+G+B)/3. This computation conflicts badly with the properties of color vision, as it computes yellow to be about six times more intense than blue with the same “lightness” value (Poynton, 1997). In other words, the way that the human eye perceives colors and brightness make it incompatible with the color spaces built around hue, saturation and brightness. HSB and HLS are not useful for image computation because of the discontinuity of hue at 360°. Arithmetic mixtures of colors expressed in polar coordinates cannot be performed. Nearly all formulations of HSB and HLS involve different computations around 60° segments of the hue circle. These calculations introduce visible discontinuities in color space. HSB and HLS are therefore not useful for conveyance of accurate color information (Poynton, 1997). Still, these color spaces carry a value in the effects that can be implemented in through them.
2.2 Software image processing

In addition to the more straightforward operations outlined in Section 2.1 all software suites aimed at image processing and color grading usually have a setup of more advanced, specialized tools. The aims of such tools are for example to aid the usability of an application or give the capability to change several features at once. These tools are more or less standardized in digital image processing programs such as Photoshop (Adobe Systems, Inc, 2005) and The GIMP (The GIMP Team, 2006). They are therefore an important inspiration to the implementation of this project. Making the distinction between a simple and a complex tool can easily be criticized as very arbitrary. However, this distinction is done as it can represent a mark of the shift from spontaneously added effects to a full-fledged color grading solution. This section will explain these specific tools in more detail: Levels, Curves, Color Balance, Hue/Saturation Variation and Colorize.

Levels lets you correct the tonal range and color balance by adjusting intensity levels of image shadows, midtones, and highlights (Adobe Help Center for Photoshop CS2). The artist can control the overall contrast, gamma and dynamic range of the entire image, or she can manipulate those properties for each color channel independently (Bjorke, 2004). The notion of shadows, midtones and highlights can undoubtedly seem a bit ambiguous. Especially shadows can be confusing since we deal with literal shadow (cast shadows), for graphics arts purposes however shadows are defined as the darkest area of the image (Margulis, 1995). Similarly the lightest white areas of an image are called highlights and in-between these two extremes rests the color range called midtones. Even more intermediate tones could be defined, e.g. quartertones and three-quartertones but the generally established ranges are shadows, midtones and highlights.

Curves is a tool that let’s the user adjust the entire tonal range of an image. Curves can be used to make precise adjustments to individual color channels in an image (Adobe Help Center for Photoshop CS2). According to Margulis (1995) input-output curves are by far the most important correction tool for any color manipulation program. According to
Bjorke (2004) *curves* are useful when an artist desire more precise control than *levels* provide, or when more unusual, nonlinear effects are required. The input-output mapping of color channels in *curves* can be defined by arbitrary cubic splines, this make them a very powerful tool but Bjorke (2004) states that it is difficult to write a single algebraic expression that defines the many possible relationships. This fact makes *curves* less than ideal for an arithmetical implementation in a *pixel shader*.

![Curves dialog from GIMP, showing quite a dramatic remapping of the blue channel.](image)

*Figure 9: Curves dialog from GIMP, showing quite a dramatic remapping of the blue channel.*

*Color Balance* changes the overall mixture of colors in an image for generalized color correction. This is done by bringing out or suppressing the red, green and blue channels. Color balancing tools usually allow the modifications to be made in the shadow, midtone and highlight ranges individually.

![Color Balance dialog from GIMP.](image)

*Figure 10: Color Balance dialog from GIMP.*
*Hue/Saturation*, this tool can be identified by a number of different names, for example HSV (Hue-Saturation-Value); (Hue-Saturation-Variation), HLS and HSB. In spite of these naming differences the various models apply very similar effects. The unifying characteristic is that they represent and manipulate colors based on a model known as the HSV color space (Brinkman, 1999). A description of the concepts of *hue* and *saturation* can be found in Section 2.1.1.

**Figure 11: Hue Saturation dialog from GIMP.**

*Colorize* is another tool based on the *HSV* color model, the difference is that instead of acting on and modifying a specific range of colors it renders the picture as a grayscale image seen through a colored glass (The GIMP Documentation Team, 2006) i.e. the entire image is tinted by one global *HSV* value. Technically the only difference between *Hue-Saturation* and *Colorize* is that the *Colorize* tool do not add the value of the incoming (original) pixel, it just takes the absolute value from the slider.

**Figure 12: Colorize dialog from GIMP.**

For the *Hue/Saturation* and *Colorize* techniques to be used the color space has to be converted from RGB space to HSV and before drawn on screen converted back to RGB. It is only in HSV (alternately HSB) space that these effects can be carried out.


2.3 GPU image processing

Image Processing can be described as the creation of a new image by processing the pixels of an existing image; each pixel in the output image is computed as a function of one or several pixels in the original image (Jargstorff, 2004). Before the age of programmable GPUs, image processing was possible but tedious, the problems were that only a small number of algorithms were available and that algorithm needed a completely new render setup. The big win with programmable graphics processors is therefore that they provide a unified approach to GPU image processing, the 3D API can not read and write pixel values (Jargstorff, 2004) motivating the practice of using pixel shaders.

Image processing is performed on a GPU by using the source image as a texture and drawing a screen-aligned quadrilateral into the back buffer or another texture. A pixel shader is used to process the input image to produce the desired result in the render target (Mitchell et. al., 2003). This basic principle can be used for both the procedural approach and the lookup approach in the course of implementation.

![Figure 13: The principle for GPU image processing outlined.](image)

The core difference between the two approaches lies in the way pixels are altered within the shader. The procedural model do color transformations through arithmetic expressions while the lookup method encode the color transform in a 3D texture, the output color is then retrieved by doing a dependant texture lookup into the 3D texture.

2.3.1 The procedural approach

The International Color Consortium (ICC) specifies two ways to store color conversion profiles. The simplest way to store a conversion from one three-channel space to another is to use a 3x3 matrix. A 3x3 matrix is in this case defined as an array of nine numbers that can convert an triplet of numbers, such as (20, 30, 40), to another triplet, such as (1.83, 2.0, 3.67). In ICC profiles, the 3x3 matrix consists of the XYZ values for each of the three colorants (primaries) of the device. The profile also contains one or more numbers defining the tone curve of each colorant. The device numbers are passed through the tone curves before conversion using the 3x3 matrix (Fraser et. al., 2003).

The other way to store the conversion is in a lookup table (LUT), which is a table of numbers that lets you look up any input value and find its corresponding output value (Fraser et. al., 2003). It is easy to draw some analogies between these specifications and the approaches that are to be examined in this dissertation. Clearly the area of color grading, especially for the procedural approach comprise more operations than color.
space conversions, the key difference is still the same though namely to distinguish between techniques achieved by arithmetic operations and techniques achieved by a lookup in a pre-computed table. The procedural techniques that have been implemented in this project are:

- Grayscale
- Invert
- Gamma
- Threshold
- Sepia tone
- Levels
- Hue/Saturation
- Colorize

For a complete listing of the code for these effects, see Appendix C2, which also list the code for the LUT method. In addition to the above listed effects a number of other techniques were implemented but abandoned due to not achieving the sought effect or due to not contributing any relevant angles of approach during testing.

The Sepia tone effect listed can be described as sort of a special case technique; the results of it can most probably be achieved by combining the effects of other tools, such as Colorize and Levels. However, being a very common effect it seems interesting to include in the tests, sepia tones appear naturally in old photographs and is commonly used to style images and movies into an “old look”, e.g. in the movie O Brother, Where Art Thou? (Coen et. al., 2000). Sepia tone can be described as transforming the image into monochrome brown shades.

### 2.3.2 The lookup table approach

Tables in general are useful data structures that are covered in virtually all data-structure textbooks. A table-driven method is a scheme that allows you to look up information in a table rather than using logic statements to figure it out (McConnell, 1993).

A classical example of the use of lookup tables is to retrieve pre-calculated sine and cosine values. This used to be a common solution back when Intels 386 and 486 processors were the fastest available to consumers, today however it is definitely faster to calculate these values, this has mainly to do with the relation between processor clock speed and memory bus clock speed. In DeLoura, 2001 it is stated that in past lookup tables were good primarily for performance reasons, but that a bigger gain today is their locality. This example shows a bit of both the strength and the weakness of lookup tables.

Color operations can be classified by their dimensionality, i.e. the number of input values required to determine an output value. All color operations that can be parameterized on a single input variable can be expressed through a curve or a one-dimensional (1D) lookup table (Selan, 2005). This way, operations such as brightness, gamma and contrast can be expressed.

2 Note that the Hue/Saturation and Colorize effects are made up of three pixel shaders each, both require a conversion first to HSV color space and after the effects have been applied a conversion back to RGB color space.
By extending the method to 3 curves, alternatively 3 lookup tables, the functions can be used independently on the color channels (red, green and blue).

![RGB(0,0,0) to RGB(1,1,1) gradient]

**Figure 14:** A 1D LUT, which can be used to map a single input value to an arbitrary output value. On the right pictured as a 1D gradient texture.

One-dimensional lookup tables become inadequate when an output value can’t be determined from a single input value. For example, a simple grayscale conversion cannot be parameterized on one variable\(^3\). To do this conversion a weighted average of the three color channels are needed. Other techniques that can not be realized with 1D LUTs are the **Hue/Saturation** and **Colorize** effects, which require input from more than one color channel to determine the output.

Three-dimensional (3D) lookup tables offer a solution to the limitations of the one-dimensional table. Most color operations can be defined through the use of 3D LUTs, the simple operators, e.g. gamma and contrast are no match using this approach. In addition, more complex operations, e.g. Hue/Saturation modifications, that in an algebraic implementation require a rather complex algorithm, are expressible as a 3D LUT with no added complexity.

![3D LUT example]

**Figure 15:** The 3D identity LUT, which can be used to map an input three-vector to an arbitrary output. On the right, the specific identity LUT used in the implementation (the uneven spacing of the dimensions exist only to clarify the illustration).

It is proposed in Selan, 2005 that the 3D LUT is generated and cached as a 2D image. Unfortunately this routine is not described in any more detail. Therefore the implementation for this project use the most obvious solution, namely to store the LUT as a volume texture in the DirectDraw Surface (.dds) file format. Volume textures are an extension of the standard texture for Direct3D 9 (Microsoft Corporation, 2006).

---

\(^3\) Technically a grayscale image could be achieved by simply using one of the color channels. However, the suitability of this rudimentary solution can strongly be argued. See Section 2.1 for more on grayscale conversion.
DirectDraw Surfaces are supported by the image library DevIL, employed in the implementation.

The identity LUT in itself has not been generated either, but rather manually created. Although by an entirely numerical approach, so this is a negligible difference.

The byte order that the texture is stored in do affect which axis should represent each color.

The 3D LUT method have a number of imperfections making it suboptimal for some tasks, Selan, 2005 highlight a number of limitations to the 3D LUT approach.

- A pixel's computation must be independent of the spatial image position. Color operators that are influenced by neighboring values are not expressible in lookup-table form.

- The color transform must be reasonably continuous, as sparsely sampled data sets are ill suited to represent discontinuous transformations. If smoothly interpolating over the sampled transform grid yields unacceptable results, lookup tables are not the appropriate acceleration technique.

- The input color space must lie within a well-defined domain. An "analytically" defined brightness operator can generate valid results over the entire domain of real numbers. However, that same operator baked into a lookup table will be valid over only a limited domain.

One fact about the present implementation of the 3D LUT method that can be considered a drawback in its usability is that it is dependent on external tools. Specifically Photoshop (Adobe Systems, Inc, 2005) and the Nvidia DDS Plug-in (Nvidia Corporation, 2003) is required for the current implementation to function. It is proposed by Bjorke, 2004 that in a production pipeline using a high-level shading language, an artist or art director could define the color correction by first opening a screen capture from the game or other image source in Photoshop, then applying the correction as a new layer in Photoshop. The writer of the shader could then open the image in Photoshop and copy the corrected values and paste them as the inputs for the color-correction code. In the implementation for this dissertation this approach has been extended from copying values to exporting a 3D LUT.
Still, an even more efficient pipeline would be capable of compiling procedural techniques into LUTs on the fly. This would allow both approaches to be configurable in real-time and not dependant on third-party solutions. Since this project is mainly aimed at the technical advantages and disadvantages of the outlined approaches the usability aspect of the implementation had to unfortunately stand back.
3 Problem description and statement

3.1 Presentation

The intention of this paper is to compare different implementations of real-time color correction aimed for use in game development. These implementations can be partitioned into arithmetically based methods and methods based on lookup tables i.e. techniques that use a mathematical function to convert the color of a pixel and techniques that lookup a precomputed evaluation of an input. The ambition for this project is firstly to implement the more naïve algorithms presented in Chapter 2.1, a second ambition is to implement some of the tools described in Chapter 2.2, e.g. techniques that share the same overall capabilities as in professional applications for color correction. It is assumed that no one solution on the whole is better than the other. A focus of the analysis will therefore be to categorize the advantages of the different approaches.

- Image quality, this aspect is of course an important one. The problem is that the quality of an image is all in the subjective evaluation of the viewer. For that reason, the evaluation of this aspect could always be argued. To give a fairly objective view this work will examine image quality in a purely technical sense. This can be done by comparing the results of the different implementations to those of a professional product.

- Performance is an important attribute when a technique is aimed for use in real-time applications. This aspect can also be separated into processor and memory performance. The processor performance in this case being how much work is put on the GPU i.e. how many instructions is needed for a shader or how many fps (frame per second) are lost when applying the technique.

- Compatibility is another aspect, somewhat related to performance but distinct enough to be mentioned on its own. Compatibility see to what hardware is needed in order to implement a specific solution.

- The flexibility of a method is also an aspect to be considered. Flexibility in this dissertation will look at the issue of if and how easily different effects can be mixed. A problem that might arise is that this also touches on subjects close to the performance aspect since combining various effects can be assumed to have a negative effect on performance.

3.2 Motivation

Five key reasons will be outlined in this section as to why color correction can be useful for game development. One of these is to use the effects to aid the artistic direction, this way can in itself be divided into two tracks. First off the more traditional way of using color correction which is mostly associated with the prepress industry, i.e. the retouching of photographs in order to enhance the image or to achieve the quite ambivalent goal referred by Margulis (1995) as “matching the art”. Depending on whom you ask this goal may be interpreted into a lot of different things, in the end however the important thing to know is that the decision about look falls on the art director. The actual usefulness of these means in game development could of course be argued. In prepress the purpose is often to adjust imperfections in photographs that arose due to less than perfect conditions...
or camera settings. These minor adjustments may nevertheless be useful in matching slightly different artwork. According to Bjorke (2004) color correction can be used to combine art elements from different sources, for example to match color palettes between different video sources, or models made by different artists. It can also give a broad coherence and mood to entire parts of a game or scene without changing the underlying models and textures.

Secondly the artistic direction can be aided in an approach more close to special effects. This can be observed in the movie *Sin City* (Rodriguez et. al., 2005), where color grading is used to a significant extent to create a heavy black & white look, replicating the style of the comic that the movie is based upon. This may be one of the more extreme examples to be found but the practice of stylization is more or less employed in all visual productions today. Most of the imagery we see on television, in magazines, and in movies have undergone very careful color correction and control (Bjorke, 2004).

The third motivation for the use of color grading is like the previous one closely related to the movie industry. This principle is to use color as a storytelling device, e.g. this can be seen in the movie *Traffic* (Soderbergh, 2000). In this picture scenes taking place in Cincinnati and Washington have a blue-tinted look, scenes taking place in Mexico have an orange-brown filter applied and scenes in San Diego have a warmer color palette applied.

The fourth major reason to employ color correction in games is to improve gameplay. This is largely unexplored territory as color correction effects have not been widely available for game designers. Adjustments that have commonly been available are brightness and gamma correction settings for the user. Expanding in this direction could see the more advanced color tools presented to the user for them to practically choose whatever color palette wanted. More interesting for the industry is the possibility for game designers to use color effects, the very recent advances in available complexity makes it an unexplored area and examples are quite sparse. One can be seen in the game *F.E.A.R* (Monolith Productions, Inc., 2005) which implements a slow-motion feature, a variant of the more generally known *Matrix-effect*. This feature is accompanied with an effect that seems to heighten the contrast of the picture. This is a rather subtle case of emphasizing a state or mood in a game, a more tangible way is to replace the health-meter by tinting the screen red to indicate the physical condition of a game character. Another example is in *World of Warcraft* (Blizzard Entertainment, Inc., 2004) where, as the player character dies all screen color is transformed into a blue-green palette, much like the *colorize* effect explained in Section 2.2. A new possible use could be to guide the player by slightly altering the colors in a level in order to indicate progress toward the goal. It should be noted that to use color transformation effects for more refined gameplay purposes would probably require some kind of underlying scripting system to be effectively utilized.

The fifth and final reason found of why to bring color correction tools into a game engine is to assist the development process. According to Sweeney (2005) one of the most important objectives for the next generation (Xbox 360, PlayStation 3) is to empower artists with the best possible tools to ensure fast turnaround in game development. Sweeney (2005) goes on by stating that *Epic Games* are happy to sacrifice a little performance and memory if that gains them more headroom for creativity. This statement
may not directly apply to color correction, but the supposition can be made that all means of supporting the content creation process will be important for the future of game development. For the most part, color correction can already be achieved in games by tweaking art assets. However, individual modification of assets is a time-consuming process and with content creation becoming the number one cost in game development today all available tools to speed up the process should be utilized. Color correction for games will also give more control to the art director in matching the look within and between scenes.
4 Description of approach and methods

There are two quite clearly defined parts in the realization of this project. These are the implementation, i.e. to create a working application with the projected effects and the testing, i.e. the examination of the implementation in order to attain result in the four categories identified in the previous chapter. This chapter will describe the software environment used for the implementation, which later developed into the test framework, additionally the hardware architecture that was used will be described. Furthermore the test categories and the various methodologies for testing will be presented.

4.1 Test framework

As a framework for implementing the planned effects the OGRE rendering engine (The OGRE Team, 2006) was selected. This choice was made partly out of previous experience with the engine but primarily because of the meticulous design, usability and comprehensive documentation of the engine. In competition with OGRE as test framework were firstly the idea to build an own environment directly on Direct3D or OpenGL. This option was abandoned due to the potential risk of putting valuable time in a lot of unnecessary groundwork, or if you will; reinventing the wheel. Another alternative that was considered, and to a limited extent tested was to carry out the implementation in FXComposer (Nvidia Corporation, 2005), which could well have worked out for achieving the desired effects. The issue with FXComposer however was an uncertainty of how to later perform the testing. Being an IDE (Integrated Development Environment), FXComposer do have an advantage in terms of usability, but the inability to have full control of the program was seen as a risk that could backfire late in the project.

The implementation in OGRE was started on the CVS HEAD, version 1.1 but was later moved over to version 1.2 Release Candidate 1 (RC1) and then to RC2. The reason for not starting the implementation on a stable version of the engine is that OGRE as of version 1.1 includes a framework called the compositor. This subsection of the API is by the manual (Streeting, 2006) described as intended for implementation of full screen post-processing effects. This purpose goes hand in hand with the implementation aims of this project.

The compositor of OGRE use Rendertargets to apply post-processing effects, the reason for this is that shader can not access the frame buffer (Jargstorff, 2004), the work-around is therefore rendering the scene into a texture (Rendertarget) which then is used as input for another rendering pass in which the post-process effect can be applied.

4.2 Test setup

Two different test architectures have been used in the course of the project; these differ in hardware specifications while effort has been made to keep the software setups equivalent. Both architectures use Windows XP Pro Service Pack 2 as operating system,
both systems have used the compilers Visual C++ 7.1 and Cg Compiler 1.4.1, the underlying API in both setups is Direct3D 9c.

The hardware for test setup 1 is a Pentium 4 processor at 3,0 GHz, a 6600GT (NV43) graphics card with 128MB memory, a GPU clock frequency of 500 MHz and a memory clock frequency of 1000 MHz, the test setup uses the ForceWare 81.98 driver. The hardware for test setup 2 is a Pentium 3 processor at 800 MHz, a 9600Pro (RV350) graphics card with 128MB memory, a GPU clock frequency of 300 MHz and a memory clock frequency of 600 MHz, the test setup uses the Catalyst 6.4 driver. The test cases for reviewing image quality have been rendered on test setup 1. The rendering for all test cases is done at resolution 1024x768 pixels.

4.3 Test procedures

Testing of image quality will be done with the tool The Compressonator (ATI Technologies, Inc., 2006), this is actually a tool for compressing textures and creating mip-map levels. The Compressonator also includes a range of means to compare image quality, meant for measuring the loss in quality as textures are compressed. These tools can also be used to evaluate the degradation of image quality as a less accurate method of realizing a particular effect is used. Specifically in this case when a lookup table is used to reproduce a procedural effect or a reference image created with an image editing application. In other words, the premise is that a procedural approach can achieve exactly the result wanted whereas the LUT give an approximation of that result. The specific statistics that can be acquired from The Compressonator are:

- Mean Error
- Error Deviation
- Square Error
- Weighted Square Error
- db (square error)
- Maximum Difference

This test will focus on the square error, which is regarded the most significant and descriptive of the listed measures when it comes to overall image quality. The error can be described here as the amount by which the estimator differs from the quantity to be estimated. In other words, assumed that the procedural method is correctly implemented, the technical assessment of image quality in the LUT method is the ability to produce an error value as low possible.

Testing the performance of each implementation is probably the most straight-forward part of the evaluation. The customary way to benchmark rendering performance is to measure the frame rate of the application. The issue consequently comes down to choosing the most appropriate way of measuring. Three ways of carrying out these tests were considered and in the end a hybrid approach was chosen. By default (due to being an extension of a previous demo) the application includes a frame rate (fps) counter. For testing however, this method can not be considered appropriate, firstly the fps counter is part of the GUI which in itself uses rendering resources, secondly it is simply to inaccurate obtain a value from this on-screen counter even though it has a computed average. Another method would be to log the frame rate at every frame over a specified
interval and then compute the total average. A third approach would be to use an external application for these objectives. The selected approach was ultimately to combine the second and third method. This might be deemed redundant, there are however two reasons for deciding on this twofold test approach. Firstly, it could not be assured that the internal benchmark implementation was 100% accurate. Therefore the external benchmark, Fraps (Beepa Pty Ltd, 2005) could serve as a verification of the internal benchmark. Secondly, the reason for not exclusively using the external benchmark was that it required to be started manually; the benchmark could consequently not be guaranteed to be consistent for all tests. The performance category will also be examined with regard to the memory demands of the implementations by computing the memory needed to allocate required resources.

The compatibility testing will mainly be performed by finding out the minimum required shader model for all techniques. It is recognized that this may not be the optimal way of verifying the hardware requirements. It would however be very hard to carry out a complete and structuralized testing of compatibility, since this would require a lot more computer hardware (primarily graphics cards) than was available in the course of the project. Another assumption worth of notice is that the compatibility may very well depend on the way that the implementation is done.

An assumption about flexibility, i.e. the ability to mix different effect is that it may depend on the manner of the implementation. To get an idea about flexibility the first step is therefore attempting to get multiple techniques running at the same time. From that point a summary of how many effects are applicable will be made, also included will be some discussion on what effects the mixing might have on other factors, such as image quality and performance.
5 Results

5.1 Image Quality

Early on in the testing of image quality it was realized that the statistics acquired really meant nothing on their own. In order to explain the maximum square error a reference test case was designed, this test case is included as the leftmost column in Figure 17 below. The reference case compares a black image with a white image, i.e. one image where all pixels have the value RGB(0,0,0) and one image where all values are RGB(255,255,255). Additionally, the reference minimum square error is zero, a result that occurs when two exactly identical images are compared.

This section primarily discusses image quality in the LUT method as avoiding falloff from the procedural and reference cases as these are considered complete in their quality, presumed that they are designed and implemented correctly.

![Image Quality (with reference)](image)

**Figure 17**: Full image quality results, this diagram is mostly useful as a reference. The leftmost test case shows difference between an entirely black and an entirely white image, i.e. the maximum possible deviation.

In the case of comparatively basic transformations or when no effect has been applied the higher precision LUT (32x32x32) can surprisingly be seen (Figure 18) producing less accurate results than the lower precision LUT (4x4x4). This result is assumed to appear as a consequence of differentiating procedures in filtering as the LUT is scaled up in size. The gain of a larger LUT comes when the effects become more complex and more substantial. This can be seen on a small-scale in the moderate levels test cases, both when comparing the procedural method to the LUT and when comparing the LUT to a Photoshop reference.
Analyzing the cases further shows that the smaller LUTs become almost unusable for correctly representing some extensive effects; the basis for this assessment is the test cases labeled extreme where acute differences occur between the images, most outstanding is the extreme levels test case. In Figure 19 parts of the tested screenshots can be viewed in order to get an idea of the image differences.

Figure 18: Image quality results without the reference.

For the complete presentation of all image quality test cases see Appendix A.

Figure 19: Screenshots of the extreme levels test case. From left to right: Original image, 128x128x128 LUT, 32x32x32 LUT, 4x4x4 LUT. Note that the images have been both reduced in size and cropped.
5.2 Performance

5.2.1 Frame rate performance

Directly notable is that in Test setup 2 of Figure 22 no results at all have been obtained from the techniques Threshold, Hue/Saturation and Colorize. This result is a consequence of hardware limitations, i.e. the Cg program code is not compatible with the GPU, for a more involved discussion on this matter see Section 5.3.

Another notable fact is that there is quite a difference in performance between the various techniques. Grayscale conversion, the least demanding algorithm, cause the fps to drop to 87% for test setup 1 and to 85,6% for test setup 2 whereas Hue/Saturation, the most demanding algorithm causes test setup 1 to drop to only 24,7% of the original frame rate (it can be assumed that any fall-back compatible with test setup 2 would achieve no better results).
Moreover it is important to watch the results achieved by the LUT method (tested for three different sampling sizes). While not ever being the fastest method overall the LUTs reach far better frame rates than for example *Hue/Saturation* and *Colorize*, with the capability of realizing these same effects. An essential fact about the LUT method is that the effect applied does not influence the frame rate at all, only the size of the LUT affects the frame rate performance.

It must be stated that a number of the shaders have some very obvious optimizations, e.g. the *Gamma* and *Levels* techniques execute the same code four times over, once for every separate color channel (red, green, blue) and once for the total pixel value. Assume a situation where one wants for example only make adjustments to the red channel, in this case \( \frac{3}{4} \) of the code would be run redundantly and can thus be stripped out.

During performance testing the minimum and maximum frame rates of the cases were gathered. As can easily be seen that these statistics are not included in the report. Firstly because they were not considered significant to the results, the way that most of these color transformations are implemented mean that they do not affect the frame rate differently over time, i.e. they execute the same code every frame.\(^5\) Secondly because variations did take place and these were quite drastic too, however these fluctuations only occurred during the very beginning of the test cases (frame \(1 \sim 300\) and can be attributed

\(^5\) Some exceptions exist here, namely the techniques using dynamic branching.
to resource loading being done by the engine. It should be emphasized that these initial variations in frame rate have been excluded from the calculated average.

5.2.2 Memory usage
The subject of performance has to be examined from two perspectives. While the frame rate performance (speed) is arguably the most important, memory usage is not of negligible interest. The issue of memory consumption is here reduced only apply to the LUT approach since the procedural implementations only ever requires a few bytes of memory, in fact the number is to small to determine a reliable test approach for the procedural implementations. For the lookup tables the amount and range of memory use is more significant and thus more interesting to address. Selan, 2005 states that a 32x32x32 LUT necessitates 393 KB of memory, this figure is unfortunately not broken down in much detail. Apart from the obvious aspect of size, i.e. LUT sampling precision the memory consumption is influenced by a number of factors. One factor is the texture format employed; another factor whether mipmapsc have been generated, a third is the potential overhead in memory use by the rendering engine. The baseline of memory use however is the size of the LUT. The layout of the LUT, i.e. being three-dimensional means that one step up in precision implies eight times the memory use. Assuming a 32-bit texture format, e.g. RGBA8 (a very common texture format) and no mipmap the theoretical memory use is the number of pixels per axis multiplied with each other; this number is then multiplied with the pixel depth (32 bits). Figure 23 depicts the memory use for different LUT sizes in more detail.

<table>
<thead>
<tr>
<th>LUT size (pixels)</th>
<th>Memory use (B)</th>
<th>Memory use (KB)</th>
<th>Memory use (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 x 4 x 4</td>
<td>256</td>
<td>0.25</td>
<td>0.000244141</td>
</tr>
<tr>
<td>8 x 8 x 8</td>
<td>2048</td>
<td>2</td>
<td>0.001953125</td>
</tr>
<tr>
<td>16 x 16 x 16</td>
<td>16384</td>
<td>16</td>
<td>0.15625</td>
</tr>
<tr>
<td>32 x 32 x 32</td>
<td>131072</td>
<td>128</td>
<td>0.125</td>
</tr>
<tr>
<td>64 x 64 x 64</td>
<td>1048576</td>
<td>1024</td>
<td>1</td>
</tr>
<tr>
<td>128 x 128 x 128</td>
<td>8388608</td>
<td>8192</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 23: LUT memory usage, all cases assume a 32-bit texture format.

5.3 Compatibility
The main discovery about the compatibility of any particular technique comes down to the complexity of the technique in question. Often the way that the implementation is done play into the compatibility of the technique, optimizations are frequently possible already to a high-level language such as Cg, and at assembly level even more refinements might be possible.

---

6 It can be stated that in the implementation of this project no gain could be found in having mipmap for the LUT.
It however becomes a problem in judging the compatibility that it depends quite heavily on the manner of the implementation, it can in addition be assumed that the test framework is not the least demanding solution, e.g. rendering to textures have been technically possible since DirectX 6 and practically feasible since DirectX 7 (Pallister, 1999), however the test implementation only supports rendering with DirectX 9 (besides as OpenGL). Another problem in estimating the compatibility is that the compiler (and the version of the compiler) might affect what hardware is required to run a shader. As mentioned in Section 5.2 some of the techniques are not able to run on test setup 2, which contains older hardware than test setup 1. Specifically this is due to use of dynamic branching, i.e. flow control instructions, a feature of shader model 3.0 (Microsoft Corporation, 2006), also to some extent featured in pixel shader 2.x a superset of pixel shader 2.0. Usually flow control can be substituted with arithmetic instructions which would bring all techniques to be compatible with a lesser level of hardware. However no manual attempts to write such fall-back techniques have been attempted in the implementation of this project. It can be noted though that all techniques in Figure 24 passes the compiler (Cg 1.4.1) using the profile pixel shader 2.0, probably by the compiler trying to convert conditional statements into arithmetic operations. The compiled assembly code is however rejected as it reaches Direct3D.

5.4 Flexibility

The flexibility that an approach holds is heavily linked to the style of implementation. Fortunately enough in this project the framework (particularly the compositor) solved this problem even before it became problem. This means that a virtually infinite number of effects can be applied, the limitation rather lie in the GPU processing capacity, as shown in Section 5.1 some effects have a grave impact on performance and applying several effects in sequence only increase this result.

---

7 Under the condition that the shader program still keeps within the instruction limit.
Figure 25: Mixing of procedural effect and LUTs. In this case leading to image artifacts (the black dots).

One prospect that was not considered in the planning of the project and therefore neither in the test cases was the possibility of implementing a hybrid approach, i.e. being able to use both procedural methods and LUTs at the same time, again though the implementation environment allowed this coincidentally. It should however be noted that this way of mixing effect can lead to unexpected degradations of image quality, e.g. Figure 25 shows the combined use of a procedural HSV effect and a 3D LUT, which results in some quite noticeable artifacts.

It must be noted about the LUT method that given a table of sufficient size one precomputed LUT can achieve the same effects as an array of precomputed LUTs that are mixed together in the rendering process. This aspect has not been tested very thorough but the test cases that were set up indicated no considerable loss in image quality while still maintaining the reasonably good performance of a single LUT.
6 Related work

It has been quite hard to find relevant related works due the course of this project. There certainly exists a research field exploring new ventures undertakings in color grading and color correction. The majority of work being done is however software based methods, i.e. rendered offline. Also much of the current work lie quite beyond the point of the implementation performed in this project, e.g. with applications such as automatically color correcting an entire movie based on adjustments made to one frame and applications applying color to black-and-white photos with a minimum of human input.

As for color grading in games the field is more restricted, and nothing else should probably be expected bearing in mind that these effects have only been realizable for the last three to four years. The prime motive of this dissertation has been to do a comparison between two different approaches to achieve color grading in real-time environments, the reason for this was that no such evaluation could be found. Much of the work done do not recognize the LUT method, e.g. Jargstorff (2004) and Mitchell et. al. (2003), rather focusing on presenting the area as an interesting undertaking for the game industry. On the other end of the spectrum stands the work of Selan (2005), a paper which is written from the preliminary perspective that LUTs are the better alternative. It can be noted that the work of Selan (2005) is not narrowed to apply only to game application but rather to all uses where real-time results are sought.

Somewhere in between the two ends of viewing the matter the work of Bjorke (2004) can be placed. In this article both ways are presented, though only 1D LUTs instead of 3D LUTs and naturally for the procedural approach; a limited number of effects. The LUT method is presented as the preferred alternative for achieving some color transformations. Bjorke (2004) also make an important observation on the probably biggest drawback of the LUT method, namely that the limitation is the lack of common tools for creating RGB-to-RGB 3D texture maps.
7 Conclusions

7.1 Summary of results

The initial hypothesis that none of the approaches would become frontrunner across the board can more or less be acknowledged as correct. However for many of the test categories the results point towards an advantage for the LUT method.

Image quality is one of the more significant test categories as it is at the core of what 3D rendering is about, namely to achieve good graphics. Although image quality of course is a matter subjective evaluation the test cases were designed from the perspective that the procedural methods could achieve ideal results and that the LUT would be an approximation. This point of view is considered valid as a manually programmed method always can achieve the desired effect given that it is correctly designed and implemented.

Image quality however is of no importance if the techniques used are not realistically achievable in real-time. Therefore the performance is of great importance when deciding on a way to implement real-time color grading. Performance is however not easy to summarize in a consistent way. While some procedural techniques perform excellent, e.g. grayscale, invert, some can almost directly be excluded from practical real-time, e.g. Hue/Saturation. This is where the LUT method comes in as the preferred alternative, an important feature of the LUT is that it achieves a constant frame rate performance regardless of what color adjustment is applied. A slight performance drop can be seen when the LUT is sized up to cope with more complicated color transforms but this is nothing compared to the fluctuations experienced with the various procedural techniques.

As for compatibility, the procedural approach come off with a mixed impression, while some effects work already with the most basic pixel shader profile (1.1) some techniques demand a more advanced profile than the LUT approach which sit solid on pixel shader 2.0.

Flexibility is probably the easiest category to analyze and can from a lazy standpoint be summarized with one word; yes. Making a more involved estimation on the subject it can be said that the level of flexibility basically is a result of the implementation. The reason for defining this category from the beginning was an uncertainty on how the implementation would be done and what it could later achieve. In an adequately capable implementation, or as in the case examined, an implementation in a capable rendering engine the issue of flexibility become a non-issue.

The final conclusion that can be made is that the choice of a proper color grading technique has to be made with consideration to the factors of what effects are sought and what costs can be paid to achieve these effects. Compromises are likely to take place but in a wide perspective the entire area of real-time rendering can be said to deal with compromises. Hopefully this dissertation has made the choice a bit easier to make.
7.2 Discussion

When the work started on this dissertation, the outlook was that the procedural way and lookup table method were two totally opposite paradigms. In the back-mirror this simple way of viewing the matter may have not been totally accurate, both approaches are most easily achieved through the use of high-level shader programs, the approaches can even combined in a hybrid approach to color grading (see Section 5.4). Nevertheless, no structuralized testing and comparison could be found on the advantages and disadvantages of the both methods, consequently this became the focus of the dissertation.

As for the two arguably most important test approaches, image quality and performance, the methodology is considered the most suitable, these test cases were also the ones that was given the largest amount of time. Still the testing was not totally inclusive, the reason for this was the hypothesis that the chosen test cases were the ones best suited for elucidating differences in all test categories. Note though that many possible test cases were excluded by the on the grounds that differences between implementations would mean that some effects could not have achieved the exact same result and are were therefore not considered appropriate for evaluation purposes. The issue of differing implementations is also the cause for doing some tests between Photoshop and LUT color adjustment instead of only comparing LUTs to procedural implementations. This is not considered a drawback to the reliability of the test cases, if the procedural techniques that were achieved could be assembled into a LUT it is safe to assume that the same amount of drift in image quality would occur.

As mentioned about compatibility in Section 5.3 a fully comprehensive test case would have required a lot more hardware than could be amassed in the scope of the project. One route to test compatibility further would have been to port the shaders to different engine, however no attempt at doing this was made as the task could have proven far to time consuming for the project. To get some sense of how the implemented techniques would translate some testing was made in OpenGL instead of Direct3D 9. The results from this examination can be described as moderately successful, even though the Cg language in theory is totally API independent the common experience is that it is seldom fully relieved of problems to go with two APIs. Some problems related to differences in matrix ordering and byte-order for the volume textures did occur. On the whole however most shaders seem to work directly and correctly, casting some hope on the portability of the implementation.

One influential limitation that was initially put on the work was the initial requirement to implement techniques that would achieve the same or nearly the same quality as professional production tools. This requirement was later revised to a desired goal. Still, a standing idea of this project is that in order to gain a practical use, the described effect need to be presented through a user interface where direct parallels can be drawn with tools that the user has seen beforehand. To achieve the same result as Photoshop is more or less impossible as the program is closed source, therefore no technical investigation can be made that ensures equality of the methods. The GIMP is an open source project so in that case an examination can be made. However, the implementations differ in quite a big way from the The GIMP’s C code with no online rendering demands to a pixel shader approach in which the entire usability stands on the real-time abilities of the implementation. Another big difference is that to optimally be able to determine the
compatibility of the methods the pixel shaders need to be compiled for as low a target shader model as possible. This means that if it is possible to realize an effect through a shader without using dynamic branching this should be done. Without the ability to use conditional statements and loops the pixel shader code has to differ quite substantially from the C code of GIMP (The GIMP Team, 2006). Therefore an initial idea to base the procedural method on the code of GIMP was abandoned.

7.3 Future work

One topic that is not included is the ability to do local color grading. This has been deliberately kept out of the project firstly to keep the implementation simpler and to keep the focus of the dissertation on the comparison between the two tested methodologies. Additionally, no easy possibilities to transfer the work routine from typical non-interactive applications could be found. Typically color grading is done in an environment where it is known exactly how every frame looks. This fact does not at all transfer to the interactive environment that a game is and consequently the work can not be performed in the same manner. A possible approach that was considered but never tried for the implementation was being able to apply effect for example to individual objects or individual render queues.

To be able to get full usability out of the implemented techniques there is usually a need for a range helping tools e.g. Histogram and Colorpicker, info on these can be found in The GIMP user manual by The GIMP Documentation Team (2006). For constructing a fully capable production environment with color grading effect these tools, or tools similar to these are required in order to aid the work process.

One approach to testing the compatibility and portability that was considered but left out due to time constraints was a sort of acid test where all techniques would be ported to another rendering engine. This experiment is quite interesting as it in addition to further examining the compatibility would give an answer on how easily transferable the implemented techniques really are. Seeing as all techniques are implemented through pixel shaders, they are assumed to be applicable elsewhere but without testing this assumption do not yet carry a great amount credence.

Moving on from the improvements that can be made to the implementation and method we can instead take the leap over to the arguably more important topic of how the described techniques can be utilized in game design. Being a dissertation in the area of computer science this work from the outset steered clear from that view of the area. Judging from the related work at least from an academic point of view the use of color grading in game design is an area where practically no research has been performed. The subject was touched on briefly in Section 3.2, although this was done without dedicating any deeper thought to the matter.
References


References


## Appendix A – Image quality statistics

<table>
<thead>
<tr>
<th>Black/White reference</th>
<th>Metric</th>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
<th>All</th>
<th>All weighted</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mean error</td>
<td>255.0</td>
<td>255.0</td>
<td>255.0</td>
<td>441.67</td>
<td>226136.55</td>
</tr>
<tr>
<td></td>
<td>Error deviation</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>Square error</td>
<td>255.0</td>
<td>255.0</td>
<td>255.0</td>
<td>3</td>
<td>255.0</td>
</tr>
<tr>
<td></td>
<td>Weighted square error</td>
<td>3</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>175.437</td>
</tr>
<tr>
<td></td>
<td>db (square error)</td>
<td>1</td>
<td>48.131</td>
<td>1</td>
<td>52.902</td>
<td>48.131</td>
</tr>
<tr>
<td></td>
<td>Max difference</td>
<td>255.0</td>
<td>255.0</td>
<td>255.0</td>
<td>255.0</td>
<td></td>
</tr>
<tr>
<td>Identity - Proc/LUT4</td>
<td>Mean error</td>
<td>-0.004</td>
<td>-0.004</td>
<td>0.0</td>
<td>0.005</td>
<td>3.136</td>
</tr>
<tr>
<td></td>
<td>Error deviation</td>
<td>0.331</td>
<td>0.296</td>
<td>0.075</td>
<td>0.451</td>
<td>0.250</td>
</tr>
<tr>
<td></td>
<td>Square error</td>
<td>4.431</td>
<td>3.969</td>
<td>4.473</td>
<td>7.441</td>
<td>4.074</td>
</tr>
<tr>
<td></td>
<td>Weighted square error</td>
<td>1.02</td>
<td>0.181</td>
<td>0.006</td>
<td>0.208</td>
<td>0.208</td>
</tr>
<tr>
<td></td>
<td>db (square error)</td>
<td>9.598</td>
<td>10.562</td>
<td>1</td>
<td>-6.921</td>
<td>-10.405</td>
</tr>
<tr>
<td></td>
<td>Max difference</td>
<td>29.0</td>
<td>10.0</td>
<td>15.0</td>
<td>29.0</td>
<td></td>
</tr>
<tr>
<td>Identity - Proc/LUT32</td>
<td>Mean error</td>
<td>1.556</td>
<td>1.131</td>
<td>3.250</td>
<td>3.776</td>
<td>1136.159</td>
</tr>
<tr>
<td></td>
<td>Error deviation</td>
<td>4.149</td>
<td>3.805</td>
<td>3.074</td>
<td>6.414</td>
<td>3.208</td>
</tr>
<tr>
<td></td>
<td>Square error</td>
<td>4.431</td>
<td>3.969</td>
<td>4.473</td>
<td>7.441</td>
<td>4.074</td>
</tr>
<tr>
<td></td>
<td>Weighted square error</td>
<td>1.367</td>
<td>2.419</td>
<td>0.367</td>
<td>2.803</td>
<td>2.803</td>
</tr>
<tr>
<td></td>
<td>db (square error)</td>
<td>11.974</td>
<td>2</td>
<td>17.435</td>
<td>12.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Max difference</td>
<td>33.0</td>
<td>12.0</td>
<td>26.0</td>
<td>33.0</td>
<td></td>
</tr>
<tr>
<td>Grayscale - Proc/LUT4</td>
<td>Mean error</td>
<td>4.348</td>
<td>4.348</td>
<td>4.348</td>
<td>7.530</td>
<td>3855.572</td>
</tr>
<tr>
<td></td>
<td>Error deviation</td>
<td>1.828</td>
<td>1.828</td>
<td>1.828</td>
<td>3.165</td>
<td>1.516</td>
</tr>
<tr>
<td></td>
<td>Square error</td>
<td>4.716</td>
<td>4.716</td>
<td>4.716</td>
<td>8.169</td>
<td>4.716</td>
</tr>
<tr>
<td></td>
<td>Weighted square error</td>
<td>1.455</td>
<td>2.874</td>
<td>0.387</td>
<td>3.245</td>
<td>3.245</td>
</tr>
<tr>
<td></td>
<td>Max difference</td>
<td>27.0</td>
<td>27.0</td>
<td>27.0</td>
<td>27.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Error deviation</td>
<td>3.266</td>
<td>3.266</td>
<td>3.266</td>
<td>5.657</td>
<td>2.709</td>
</tr>
<tr>
<td></td>
<td>Square error</td>
<td>5.456</td>
<td>5.456</td>
<td>5.456</td>
<td>9.450</td>
<td>5.456</td>
</tr>
<tr>
<td></td>
<td>Weighted square error</td>
<td>1.684</td>
<td>3.325</td>
<td>0.447</td>
<td>3.754</td>
<td>3.754</td>
</tr>
<tr>
<td></td>
<td>db (square error)</td>
<td>14.73</td>
<td>14.738</td>
<td>14.73</td>
<td>19.509</td>
<td>14.738</td>
</tr>
</tbody>
</table>
### Appendix A – Image quality statistics

<table>
<thead>
<tr>
<th></th>
<th>Max difference</th>
<th>8</th>
<th>27.0</th>
<th>8</th>
<th>27.0</th>
<th>8</th>
<th>27.0</th>
<th>8</th>
<th>27.0</th>
<th>8</th>
<th>27.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Grayscale - Proc/Photoshop</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean error</td>
<td>2.797</td>
<td>2.797</td>
<td>2.797</td>
<td>4.844</td>
<td>2479.980</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error deviation</td>
<td>1.748</td>
<td>1.748</td>
<td>1.748</td>
<td>3.028</td>
<td>1.450</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Square error</td>
<td>3.298</td>
<td>3.298</td>
<td>3.298</td>
<td>5.712</td>
<td>3.298</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weighted square error</td>
<td>1.018</td>
<td>2.010</td>
<td>0.270</td>
<td>2.269</td>
<td>2.269</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>db (square error)</td>
<td>5</td>
<td>10.365</td>
<td>5</td>
<td>15.136</td>
<td>10.365</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max difference</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Grayscale - Proc/GIMP</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean error</td>
<td>-</td>
<td>-1.700</td>
<td>-1.700</td>
<td>2.945</td>
<td>1507.884</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error deviation</td>
<td>2.626</td>
<td>2.626</td>
<td>2.626</td>
<td>4.548</td>
<td>2.178</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Square error</td>
<td>3.128</td>
<td>3.128</td>
<td>3.128</td>
<td>5.418</td>
<td>3.128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weighted square error</td>
<td>0.965</td>
<td>1.906</td>
<td>0.256</td>
<td>2.152</td>
<td>2.152</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max difference</td>
<td>4.0</td>
<td>4.0</td>
<td>4.0</td>
<td>4.0</td>
<td>4.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Invert - Proc/LUT4</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean error</td>
<td>0.004</td>
<td>0.004</td>
<td>0.0</td>
<td>0.005</td>
<td>3.450</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error deviation</td>
<td>0.332</td>
<td>0.305</td>
<td>0.075</td>
<td>0.456</td>
<td>0.256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Square error</td>
<td>0.332</td>
<td>0.305</td>
<td>0.075</td>
<td>0.457</td>
<td>0.308</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weighted square error</td>
<td>0.102</td>
<td>0.186</td>
<td>0.006</td>
<td>0.212</td>
<td>0.212</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>db (square error)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>22.51</td>
<td>10.220</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max difference</td>
<td>49.0</td>
<td>69.0</td>
<td>1.0</td>
<td>69.0</td>
<td>49.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Invert - Proc/LUT32</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean error</td>
<td>-</td>
<td>-1.046</td>
<td>3.221</td>
<td>3.675</td>
<td>1055.288</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error deviation</td>
<td>4.147</td>
<td>3.8</td>
<td>3.066</td>
<td>6.406</td>
<td>3.204</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Square error</td>
<td>4.386</td>
<td>3.942</td>
<td>4.447</td>
<td>7.386</td>
<td>4.042</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weighted square error</td>
<td>1.353</td>
<td>2.402</td>
<td>0.365</td>
<td>2.781</td>
<td>2.781</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>db (square error)</td>
<td>1</td>
<td>11.914</td>
<td>1</td>
<td>17.368</td>
<td>12.133</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max difference</td>
<td>52.0</td>
<td>72.0</td>
<td>5.0</td>
<td>72.0</td>
<td>52.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Levels (moderate) - Proc/LUT4</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean error</td>
<td>0.928</td>
<td>-0.947</td>
<td>2.676</td>
<td>2.986</td>
<td>877.128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error deviation</td>
<td>4.387</td>
<td>2.482</td>
<td>12.02</td>
<td>13.038</td>
<td>2.721</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Square error</td>
<td>4.485</td>
<td>2.657</td>
<td>4</td>
<td>13.375</td>
<td>3.426</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weighted square error</td>
<td>1.384</td>
<td>1.619</td>
<td>1.010</td>
<td>2.357</td>
<td>2.357</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>db (square error)</td>
<td>13.03</td>
<td>21.81</td>
<td>22.526</td>
<td>10.697</td>
<td>10.697</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max difference</td>
<td>35.0</td>
<td>50.0</td>
<td>59.0</td>
<td>59.0</td>
<td>59.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Levels (moderate) - Proc/LUT128

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean error</td>
<td>1.940</td>
<td>1.476</td>
<td>6.612</td>
<td>7.047</td>
<td>1558.145</td>
</tr>
<tr>
<td>Square error</td>
<td>5.150</td>
<td>4.493</td>
<td>5</td>
<td>13.476</td>
<td>4.805</td>
</tr>
<tr>
<td>Weighted square error</td>
<td>1.589</td>
<td>2.738</td>
<td>0.952</td>
<td>3.306</td>
<td>3.306</td>
</tr>
<tr>
<td>db (square error)</td>
<td>6</td>
<td>13.050</td>
<td>0</td>
<td>22.591</td>
<td>13.634</td>
</tr>
<tr>
<td>Max difference</td>
<td>35.0</td>
<td>51.0</td>
<td>61.0</td>
<td>61.0</td>
<td></td>
</tr>
</tbody>
</table>

### Levels (moderate) - Photoshop/LUT4

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean error</td>
<td>0.102</td>
<td>0.025</td>
<td>5.018</td>
<td>5.019</td>
<td>532.284</td>
</tr>
<tr>
<td>Error deviation</td>
<td>1.115</td>
<td>2.724</td>
<td>9.240</td>
<td>9.703</td>
<td>2.250</td>
</tr>
<tr>
<td>Square error</td>
<td>1.120</td>
<td>2.742</td>
<td>5</td>
<td>10.924</td>
<td>2.779</td>
</tr>
<tr>
<td>Weighted square error</td>
<td>0.346</td>
<td>1.671</td>
<td>0.862</td>
<td>1.912</td>
<td>1.912</td>
</tr>
<tr>
<td>db (square error)</td>
<td>0.985</td>
<td>8.761</td>
<td>6</td>
<td>20.768</td>
<td>8.877</td>
</tr>
<tr>
<td>Max difference</td>
<td>35.0</td>
<td>50.0</td>
<td>62.0</td>
<td>62.0</td>
<td></td>
</tr>
</tbody>
</table>

### Levels (moderate) - Photoshop/LUT32

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean error</td>
<td>1.918</td>
<td>-1.421</td>
<td>5.783</td>
<td>6.256</td>
<td>1483.621</td>
</tr>
<tr>
<td>Error deviation</td>
<td>4.764</td>
<td>4.876</td>
<td>5.921</td>
<td>10.288</td>
<td>2.278</td>
</tr>
<tr>
<td>Square error</td>
<td>5.135</td>
<td>5.078</td>
<td>8.276</td>
<td>10.984</td>
<td>5.149</td>
</tr>
<tr>
<td>Weighted square error</td>
<td>1.585</td>
<td>3.095</td>
<td>0.679</td>
<td>3.543</td>
<td>3.543</td>
</tr>
<tr>
<td>db (square error)</td>
<td>14.21</td>
<td>18.35</td>
<td>1.040</td>
<td>14.235</td>
<td></td>
</tr>
<tr>
<td>Max difference</td>
<td>59.0</td>
<td>84.0</td>
<td>28.0</td>
<td>84.0</td>
<td></td>
</tr>
</tbody>
</table>

### Levels (moderate) - Photoshop/LUT128

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean error</td>
<td>0.081</td>
<td>0.031</td>
<td>4.004</td>
<td>4.005</td>
<td>425.084</td>
</tr>
<tr>
<td>Error deviation</td>
<td>1.106</td>
<td>1.352</td>
<td>5.144</td>
<td>5.432</td>
<td>1.189</td>
</tr>
<tr>
<td>Square error</td>
<td>1.109</td>
<td>1.352</td>
<td>6.518</td>
<td>6.749</td>
<td>1.512</td>
</tr>
<tr>
<td>Weighted square error</td>
<td>0.342</td>
<td>0.824</td>
<td>0.535</td>
<td>1.040</td>
<td>1.040</td>
</tr>
<tr>
<td>db (square error)</td>
<td>0.898</td>
<td>2.618</td>
<td>3</td>
<td>16.584</td>
<td>3.588</td>
</tr>
<tr>
<td>Max difference</td>
<td>59.0</td>
<td>87.0</td>
<td>29.0</td>
<td>87.0</td>
<td></td>
</tr>
</tbody>
</table>

### Levels (extreme) - Photoshop/LUT4

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean error</td>
<td>4.823</td>
<td>7.723</td>
<td>3</td>
<td>15.768</td>
<td>6506.564</td>
</tr>
<tr>
<td>Error deviation</td>
<td>14.52</td>
<td>56.179</td>
<td>26.70</td>
<td>63.877</td>
<td>41.711</td>
</tr>
</tbody>
</table>

---

37
Appendix A – Image quality statistics

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Square error</strong></td>
<td>15.30</td>
<td>29.64</td>
</tr>
<tr>
<td><strong>Weighted square error</strong></td>
<td>4.722</td>
<td>23.69</td>
</tr>
<tr>
<td><strong>db (square error)</strong></td>
<td>5.000</td>
<td>23.69</td>
</tr>
<tr>
<td><strong>Max difference</strong></td>
<td>96.0</td>
<td>114.0</td>
</tr>
</tbody>
</table>

**Levels (extreme) - Photoshop/LUT32**

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mean error</strong></td>
<td>2.411</td>
<td>6.755</td>
</tr>
<tr>
<td><strong>Error deviation</strong></td>
<td>9.430</td>
<td>12.70</td>
</tr>
<tr>
<td><strong>Square error</strong></td>
<td>9.733</td>
<td>19.76</td>
</tr>
<tr>
<td><strong>Weighted square error</strong></td>
<td>3.004</td>
<td>19.76</td>
</tr>
<tr>
<td><strong>db (square error)</strong></td>
<td>5.000</td>
<td>21.558</td>
</tr>
<tr>
<td><strong>Max difference</strong></td>
<td>84.0</td>
<td>121.0</td>
</tr>
</tbody>
</table>

**Levels (extreme) - Photoshop/LUT128**

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mean error</strong></td>
<td>0.853</td>
<td>3.983</td>
</tr>
<tr>
<td><strong>Error deviation</strong></td>
<td>4.037</td>
<td>8.317</td>
</tr>
<tr>
<td><strong>Square error</strong></td>
<td>4.126</td>
<td>9.202</td>
</tr>
<tr>
<td><strong>Weighted square error</strong></td>
<td>1.273</td>
<td>12.31</td>
</tr>
<tr>
<td><strong>db (square error)</strong></td>
<td>1.000</td>
<td>20.035</td>
</tr>
<tr>
<td><strong>Max difference</strong></td>
<td>89.0</td>
<td>119.0</td>
</tr>
</tbody>
</table>

**HSV (moderate) - Photoshop/LUT4**

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mean error</strong></td>
<td>3.577</td>
<td>1.754</td>
</tr>
<tr>
<td><strong>Error deviation</strong></td>
<td>7.019</td>
<td>4.030</td>
</tr>
<tr>
<td><strong>Square error</strong></td>
<td>7.878</td>
<td>4.395</td>
</tr>
<tr>
<td><strong>Weighted square error</strong></td>
<td>2.431</td>
<td>17.92</td>
</tr>
<tr>
<td><strong>db (square error)</strong></td>
<td>9.000</td>
<td>15.521</td>
</tr>
<tr>
<td><strong>Max difference</strong></td>
<td>24.0</td>
<td>23.0</td>
</tr>
</tbody>
</table>

**HSV (moderate) - Photoshop/LUT32**

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mean error</strong></td>
<td>1.165</td>
<td>3.460</td>
</tr>
<tr>
<td><strong>Error deviation</strong></td>
<td>3.150</td>
<td>3.959</td>
</tr>
<tr>
<td><strong>Square error</strong></td>
<td>3.359</td>
<td>5.258</td>
</tr>
<tr>
<td><strong>Weighted square error</strong></td>
<td>1.036</td>
<td>10.52</td>
</tr>
<tr>
<td><strong>db (square error)</strong></td>
<td>3.000</td>
<td>12.998</td>
</tr>
<tr>
<td><strong>Max difference</strong></td>
<td>23.0</td>
<td>30.0</td>
</tr>
</tbody>
</table>

**HSV (moderate) - Photoshop/LUT128**

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mean error</strong></td>
<td>0.010</td>
<td>2.135</td>
</tr>
<tr>
<td><strong>Error deviation</strong></td>
<td>1.972</td>
<td>3.750</td>
</tr>
<tr>
<td><strong>Square error</strong></td>
<td>1.972</td>
<td>4.502</td>
</tr>
<tr>
<td><strong>Weighted square</strong></td>
<td>0.609</td>
<td>1.152</td>
</tr>
</tbody>
</table>

38
### HSV (extreme) - Photoshop/LUT4

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Mean Error</th>
<th>Error Deviation</th>
<th>Max Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean error</td>
<td>6.406</td>
<td>14.51</td>
<td>26.0</td>
</tr>
<tr>
<td>Error deviation</td>
<td>5</td>
<td>15.86</td>
<td>11.48</td>
</tr>
<tr>
<td>Square error</td>
<td>6</td>
<td>25.632</td>
<td>0</td>
</tr>
<tr>
<td>Weighted square error</td>
<td>4.896</td>
<td>24.00</td>
<td>11.48</td>
</tr>
<tr>
<td>Max difference</td>
<td>31.0</td>
<td>26.0</td>
<td>12.0</td>
</tr>
</tbody>
</table>

### HSV (extreme) - Photoshop/LUT32

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Mean Error</th>
<th>Error Deviation</th>
<th>Max Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean error</td>
<td>1.404</td>
<td>7.615</td>
<td>49.0</td>
</tr>
<tr>
<td>Error deviation</td>
<td>7.744</td>
<td>16.440</td>
<td>11.48</td>
</tr>
<tr>
<td>Square error</td>
<td>2.390</td>
<td>17.77</td>
<td>31.0</td>
</tr>
<tr>
<td>Weighted square error</td>
<td>9</td>
<td>24.318</td>
<td>11.48</td>
</tr>
<tr>
<td>Max difference</td>
<td>63.0</td>
<td>79.0</td>
<td>12.0</td>
</tr>
</tbody>
</table>

### HSV (extreme) - Photoshop/LUT128

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Mean Error</th>
<th>Error Deviation</th>
<th>Max Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean error</td>
<td>1.477</td>
<td>6.208</td>
<td>39.0</td>
</tr>
<tr>
<td>Error deviation</td>
<td>6.381</td>
<td>8.756</td>
<td>29.0</td>
</tr>
<tr>
<td>Square error</td>
<td>1.969</td>
<td>16.09</td>
<td>31.0</td>
</tr>
<tr>
<td>Weighted square error</td>
<td>8</td>
<td>19.713</td>
<td>11.48</td>
</tr>
<tr>
<td>Max difference</td>
<td>56.0</td>
<td>77.0</td>
<td>12.0</td>
</tr>
</tbody>
</table>
# Appendix B – Frame rate benchmarks

## Test setup 1 (6600gt)

<table>
<thead>
<tr>
<th>Technique</th>
<th>Internal benchmark</th>
<th>Fraps</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Default (no effect)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average fps:</td>
<td>281.37</td>
<td>280.847</td>
</tr>
<tr>
<td><strong>Grayscale</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average fps:</td>
<td>244.969</td>
<td>244.384</td>
</tr>
<tr>
<td><strong>Invert</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average fps:</td>
<td>242.545</td>
<td>243.626</td>
</tr>
<tr>
<td><strong>Gamma</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average fps:</td>
<td>166.829</td>
<td>166.915</td>
</tr>
<tr>
<td><strong>Threshold</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average fps:</td>
<td>238.463</td>
<td>238.590</td>
</tr>
<tr>
<td><strong>Sepia tone</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average fps:</td>
<td>221.217</td>
<td>221.363</td>
</tr>
<tr>
<td><strong>Levels</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average fps:</td>
<td>147.522</td>
<td>147.580</td>
</tr>
<tr>
<td><strong>Hue/Saturation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average fps:</td>
<td>69.4627</td>
<td>69.491</td>
</tr>
<tr>
<td><strong>Colorize</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average fps:</td>
<td>69.4952</td>
<td>69.524</td>
</tr>
<tr>
<td><strong>3D LUT (4x4x4)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average fps:</td>
<td>230.712</td>
<td>230.726</td>
</tr>
<tr>
<td><strong>3D LUT (32x32x32)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average fps:</td>
<td>223.711</td>
<td>223.783</td>
</tr>
<tr>
<td><strong>3D LUT (128x128x128)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average fps:</td>
<td>211.575</td>
<td>211.444</td>
</tr>
</tbody>
</table>
### Appendix B – Frame rate benchmarks

#### Test setup 2 (9600pro)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Internal benchmark</th>
<th>Fraps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default (no effect)</td>
<td>Average fps: 120.109</td>
<td>120.161</td>
</tr>
<tr>
<td>Grayscale</td>
<td>Average fps: 102.799</td>
<td>102.837</td>
</tr>
<tr>
<td>Invert</td>
<td>Average fps: 102.804</td>
<td>102.841</td>
</tr>
<tr>
<td><strong>Gamma</strong></td>
<td>Average fps: 57.6886</td>
<td>57.712</td>
</tr>
<tr>
<td>Threshold</td>
<td>Average fps: -</td>
<td>-</td>
</tr>
<tr>
<td>Sepia tone</td>
<td>Average fps: 84.125</td>
<td>84.156</td>
</tr>
<tr>
<td><strong>Levels</strong></td>
<td>Average fps: 49.2494</td>
<td>49.268</td>
</tr>
<tr>
<td>Hue/Saturation</td>
<td>Average fps: -</td>
<td>-</td>
</tr>
<tr>
<td><strong>Colorize</strong></td>
<td>Average fps: -</td>
<td>-</td>
</tr>
<tr>
<td><strong>3D LUT (4x4x4)</strong></td>
<td>Average fps: 94.4553</td>
<td>94.491</td>
</tr>
<tr>
<td><strong>3D LUT (32x32x32)</strong></td>
<td>Average fps: 90.0099</td>
<td>89.703</td>
</tr>
<tr>
<td><strong>3D LUT (128x128x128)</strong></td>
<td>Average fps: 81.2004</td>
<td>81.206</td>
</tr>
</tbody>
</table>
Appendix C – Cg shader code for test cases

**Grayscale**

sampler RT : register(s0);  
float4 main_fp
{
    float2 iTexCoord : TEXCOORD0,
    uniform sampler2D RT,  // Rendertarget
    uniform float3 rgbWeights  // Default=(0.222, 0.707, 0.071)
} : COLOR
{
    float4 oColor;
    float3 color = tex2D(RT, iTexCoord).rgb;  // Get the color
    oColor.rgb = dot(float3(rgbWeights.r, rgbWeights.g, rgbWeights.b),
        color);
    oColor.a = 1;
    return oColor;
}

**Invert**

dsampler RT : register(s0);  
float4 main_fp
{
    float2 iTexCoord : TEXCOORD0,
    uniform sampler2D RT  // Rendertarget
} : COLOR
{
    float4 oColor;
    float3 color = tex2D(RT, iTexCoord).rgb;  // Get the color
    oColor.rgb = 1 - color;
    oColor.a = 1;
    return oColor;
}

**Gamma**

sampler RT : register(s0);  
float4 main_fp
{
    float2 iTexCoord : TEXCOORD0,
    uniform sampler2D RT,  // Rendertarget
    uniform float4 gamma
} : COLOR
{
    float4 oColor;
    float3 color = tex2D(RT, iTexCoord).rgb;  // Get the color
    oColor.rgb = pow(color, (1/gamma.a));
    oColor.r = pow(oColor.r, 1/(gamma.r));
    oColor.g = pow(oColor.g, 1/(gamma.g));
    oColor.b = pow(oColor.b, 1/(gamma.b));
Threshold
sampler RT : register(s0);
float4 main_fp
{
    float2 iTexCoord : TEXCOORD0,
    uniform sampler2D RT, // Rendertarget
    uniform float value
} : COLOR
{
    float4 oColor = 0;
    float3 color = tex2D(RT, iTexCoord).rgb; // Get the color
    float pixelvalue = color;
    if (pixelvalue > value)
    oColor.rgb = 1;
    else
    oColor.rgb = 0;
    oColor.a = 1;
    return oColor;
}

Sepia tone
sampler RT : register(s0);
float4 main_fp
{
    float2 iTexCoord : TEXCOORD0,
    uniform sampler2D RT // Rendertarget
} : COLOR
{
    float4 oColor = 0.5;
    float4 color = tex2D(RT, iTexCoord); // Get the color
    float4 currFrameSampleYIQ;
    float4x4 YIQMatrix = { 0.299,  0.587,  0.114, 0,
                          0.596, -0.275, -0.321, 0,
                          0.212, -0.523,  0.311, 0, 0,0,0,0};
    float4x4 inverseYIQ={
                          1.0000000000000000000,  .95568806036115671171, .61985809445637075388, 0,
                          1.0000000000000000000, -.27158179694405859326, -.64687381613840131330, 0,
                          1.0000000000000000000, -1.1081773266826619523, 1.7050645599191817149, 0, 0,0,0,0 };
    // convert to YIQ space
    currFrameSampleYIQ = mul(YIQMatrix , color);
    currFrameSampleYIQ.y = 0.2;
    // convert YIQ color to sepia tone
    currFrameSampleYIQ.z = 0.0;
    // convert back to RGB
    oColor = mul( inverseYIQ, currFrameSampleYIQ);
    return oColor;
}

Appendix C – Cg shader code for test cases
Appendix C – Cg shader code for test cases

Levels

sampler RT : register(s0);

float4 main_fp
{
    float2 iTexCoord : TEXCOORD0,

    uniform sampler2D RT,       // Rendertarget
    uniform float4 inWhite,
    uniform float4 inBlack,
    uniform float4 inGamma,
    uniform float4 outWhite,
    uniform float4 outBlack
} : COLOR
{
    float4 oColor;
    float3 color = tex2D(RT, iTexCoord).rgb;   // Get the color

    oColor.rgb = (pow(((color * 255.0) - inBlack.a) / (inWhite.a -
inBlack.a), inGamma.a) * (outWhite.a - outBlack.a) + outBlack.a) / 255.0;

    oColor.r = (pow(((oColor.r * 255.0) - inBlack.r) / (inWhite.r -
inBlack.r), inGamma.r) * (outWhite.r - outBlack.r) + outBlack.r) / 255.0;
    oColor.g = (pow(((oColor.g * 255.0) - inBlack.g) / (inWhite.g -
inBlack.g), inGamma.g) * (outWhite.g - outBlack.g) + outBlack.g) / 255.0;
    oColor.b = (pow(((oColor.b * 255.0) - inBlack.b) / (inWhite.b -
inBlack.b), inGamma.b) * (outWhite.b - outBlack.b) + outBlack.b) / 255.0;

    oColor.a = 1;
    return oColor;
}

Hue/Saturation

sampler RT : register(s0);

float4 main_fp
{
    float2 iTexCoord : TEXCOORD0,

    uniform sampler2D RT,       // Rendertarget
    uniform float3 hsvValues
} : COLOR
{
    float4 oColor;
    float3 color = tex2D(RT, iTexCoord).rgb;   // Get the color

    oColor.r = abs(color.x + hsvValues.x);
    oColor.g = color.y + hsvValues.y;
    oColor.b = color.z + hsvValues.z;

    oColor.a = 1;
    return oColor;
}

Colorize

sampler RT : register(s0);

float4 main_fp
{
    float2 iTexCoord : TEXCOORD0,

    uniform sampler2D RT,       // Rendertarget

    uniform float4 inWhite,
    uniform float4 inBlack,
    uniform float4 inGamma,
    uniform float4 outWhite,
    uniform float4 outBlack
uniform float3 hsvValues
{
float4 oColor;
float3 color = tex2D(RT, iTexCoord).rgb; // Get the color

oColor.r = saturate(abs(hsvValues.x));
oColor.g = color.y + hsvValues.y;
oColor.b = color.z + hsvValues.z;

oColor.a = 1;
return oColor;
}

RGBToHSV

sampler RT : register(s0);

float4 main_fp
{
float2 iTexCoord : TEXCOORD0,
uniform sampler2D RT // Rendertarget
} : COLOR
{
float4 oColor;
float3 color = tex2D(RT, iTexCoord).rgb; // Get the color

float colorMax, colorMin;
float3 hsv;
float4 rgbd; // r, g, b, delta;

rgbd.rgb = color;

colorMax = max(rgbd.r, rgbd.g);
colorMax = max(colorMax, rgbd.b);
colorMin = min(rgbd.r, rgbd.g);
colorMin = min(colorMin, rgbd.b);
hsv.z = colorMax; // this is value
if (colorMax != 0)
{
  hsv.y = (colorMax - colorMin ) / colorMax;
}
if (hsv.y != 0)
{
   rgbd.w = colorMax - colorMin; //delta = colorMax - colorMin;
   if (rgbd.r == colorMax)
   {
     hsv.x = (rgbd.g-rgbd.b) / rgbd.w;
   }
   else if (rgbd.g == colorMax)
   {
     hsv.x = 2.0 + (rgbd.b-rgbd.r) / rgbd.w;
   }
   else // b is max
   {
     hsv.x = 4.0 + (rgbd.r-rgbd.g) / rgbd.w;
   }
}
hsv.x *= 60;
if (hsv.x < 0)
{
  hsv.x += 360;
}
oColor.r = hsv.x / 360.0;
// moving h to be between 0 and 1.
oColor.g = hsv.y;
oColor.b = hsv.z;

return oColor;

}  

HSVtoGRB 

sampler RT : register(s0);

float4 main_fp 
{
    float2 iTexCoord : TEXCOORD0,
    
    uniform sampler2D RT  // Rendertarget 
) : COLOR 
} 

{
    float4 oColor;
    float3 color = tex2D(RT, iTexCoord).rgb; // Get the color

    float i;
    float3 hsv, rgb;
    float4 fpqt;

    if (color.g == 0) 
    {
        if (color.b !=0)
        {
            oColor = color.b; // black and white case
        }
    }
    else
    {
        hsv.x = color.x * 360.0;
        hsv.y = color.y;
        hsv.z = color.z;

        if (hsv.x == 360.0)
        {
            hsv.x = 0;
        }
        hsv.x /=60;
        i = floor(hsv.x);
        fpqt.x = hsv.x - i;
        fpqt.y = hsv.z * (1.0 - hsv.y);
        fpqt.z = hsv.z * (1.0 -(hsv.y * fpqt.x));
        fpqt.w = hsv.z * (1.0 -(hsv.y * (1.0 - fpqt.x)));

        if (i == 0)
        {
            rgb.x = hsv.z;
            rgb.y = fpqt.w;
            rgb.z = fpqt.y;
        }
        else if (i== 1)
        {
            rgb.x = fpqt.z;
            rgb.y = hsv.z;
            rgb.z = fpqt.y;
        }
        else if (i== 2)
        {
            rgb.x = fpqt.y;
            rgb.y = hsv.z;
            rgb.z = fpqt.w;
        }
    }
}
} else if (i == 3) {
    rgb.x = fpqt.y;
    rgb.y = fpqt.z;
    rgb.z = hsv.z;
} else if (i == 4) {
    rgb.x = fpqt.w;
    rgb.y = fpqt.y;
    rgb.z = hsv.z;
} else if (i == 5) {
    rgb.x = hsv.z;
    rgb.y = fpqt.y;
    rgb.z = fpqt.z;
}

oColor.r = rgb.x;
oColor.g = rgb.y;
oColor.b = rgb.z;

return oColor;

3D LUT

sampler RT : register(s0);
sampler LUT : register(s1);

float4 main_fp
{
    float2 iTexCoord : TEXCOORD0,
    uniform sampler2D RT,       // Rendertarget
    uniform sampler3D LUT       // Lookup Table
} : COLOR
{
    float4 oColor;
    float3 color  = tex2D(RT, iTexCoord).rgb;  // Get the color

    // oColor = tex3D(LUT, color);
    // Correct the filtering artifacts
    oColor = tex3D(LUT, (32+192*color)/256);
    oColor.a = 1;

    return oColor;
}