A Constraint-based Approach to Dynamic Colour Management for Windowing Interfaces

by

Blair MacIntyre

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1991

Abstract

Selecting harmonious colours for traditional window systems can be a difficult and frustrating endeavor. At the root of this problem is the fact that typical window systems do not allow abstract properties of colours to be specified. Instead, they insist that users specify individual colour values exactly. When many colours are used, the value of each colour must be chosen to satisfy any relationships that exist between it and previously chosen colours. Unfortunately, the difficulty of colour selection often prevents users from taking full advantage of the functional benefits of colour, particularly that of resolving context. A more desirable approach is to allow the aesthetic and functional properties of colours to be specified and to allow users to select values for the colours they wish. The window system can choose the remaining colours using these properties. Another failing of typical window systems is that once a colour value has been determined it will not change without explicit direction from the user. When windows open or close the factors which motivated a choice of colour value may change. Unfortunately, if the user wishes the chosen colour value to change as the environment changes, he or she must typically perform the modifications. A *dynamic* window system assists the user in making these choices. By specifying colour properties as constraints, a dynamic window system can adjust colour values as the environment changes, to satisfy these constraints. The potential problems with dynamic window systems incorporating colour constraints are investigated in this thesis. An implementation that uses a distributed, jostling, constraint-solver based on a simple dynamical system shows that this approach is possible.

Acknowledgements

This work was supported by the Natural Science and Engineering Research Council of Canada through a post-graduate scholarship and grants to the University of Waterloo Computer Graphics Laboratory; the Ontario Information Technology Research Centre through grants to the laboratory; by Digital Equipment Canada through equipment donations; by Apple Computer Corporation through grants to the laboratory; and by Sun Microsystems through software support.

I would like to thank my supervisor, William Cowan, for having the patience to teach me what I needed to know about colour theory and for agreeing that this was an interesting topic. He also deserves credit for suggesting the use of a dynamical system to solve the constraints. I would also like to thank my readers, Marceli Wein, Peter Forsyth, Philippe Bertrand and Katy Simonsen for their comments. Thanks go to Maureen Stone finding those elusive CG&A articles, to Jim Lai for the OSA XYZ coordinates in Appendix A, to Josh Siegel and Don Hopkins for answering many NeWS related questions, and to all the members of CGL for providing an interesting work environment, to say the least. I would also like to thank Chris Wein for seeing this through the submission process in my absence.

A special thanks to Katy Simonsen, without whose love, support, tolerance and nods at the appropriate times I never would have made it through those last four months. I will be forever grateful.

Also, I want to thank my mother for doing everything I could ask for over the years, and my brother Craig who, among other things, introduced me to computers all those years ago.

Contents

List of Tables

List of Figures

Trademarks

X11 is a registered trademark of Massachussetts Institute of Technology. Macintosh is a registered trademark of Apple Computer. NeWS, OpenWindows and TNT are registered trademarks of Sun Microsystems Inc. Unix is a registered trademark of AT&T. PostScript is a registered trademark of Adobe Systems Inc. IRIS is a registered trademark of Silicon Graphics Inc.

All other products mentioned in this thesis are trademarks of their respective companies. The use of general descriptive names, trade names, trademarks, etc., in this publication, even if the former are not identified, is not to be taken as a sign that such names may be used freely by anyone.

Chapter 1

Introduction

> *Color is life; for a world without colors appears to us as dead.* (Itten, 1961)

> *With knowledge, an artist can take three or four unusual colors and come out with beautiful expression. It is important to study structured color schemes simply to develop this knowledge and to learn some sure-fire ways of developing beautiful color schemes and color harmony. Painting is like any other art form: Just as in dance, music, or writing, once one has a thorough understanding of all the basics of the art, the rules can be adjusted and expression becomes subjective.* (Quiller, 1989)

Until very recently, high resolution colour displays were reserved for the elite few whose work justified the expense. This has all changed. Megapixel displays, capable of displaying 256 or more simultaneous colours chosen from a palette of over 16 million colours, are becoming inexpensive and therefore more common. In addition, the computers driving these displays are running multi-tasking operating systems, allowing many programs to be run simultaneously. Each program has its own set of windows, its own user interface and its own set of colour needs.

Unfortunately, the sophistication of the window systems running on these displays is not keeping pace with the hardware. Much work is being done to make the window systems look polished; stylish three dimensional effects and flashy demo programs are common. However, tools that allow the user to take full advantage of these large colourful displays and of the powerful computers driving them have not been developed.

In particular, the tools available for making aesthetic colour choices are exceedingly poor. There are several tools that allow users to specify individual colours, but few that assist the user in coordinating all of the colours being used. Those that do address the problem are limited in scope and not generally useful.

1.1 What are users trying to do with colour?

Individual colours on a computer display are not selected in a vacuum. Each individual colour can be thought of as a piece in a much larger puzzle: that of creating a functional and attractive computer display. In order to create an environment in which the window system helps the user select colours, it is necessary to determine what people are trying to accomplish with colour and window systems.

Reasons for using colour range from the aesthetic to the functional. For example, colour is often used to create an attractive work environment, to label items, to group related elements of an interface, to signal the user about potentially dangerous actions, to enhance the realism of an image, and to view multidimensional data. Many of

these topics have been examined in depth and rules were developed to guide the application designer in deciding when and when not to use colour. Meier presents a detailed summary of much of this work (Meier, 1987).

Some decisions about the use of colour are hidden from the user. Whether related visual elements in an interface are colour coded is usually decided by the application designer. Likewise, the way in which colour is used in a data visualization or realistic rendering application is usually specified when the application is designed. In a sense, many of the decisions about colour usage within a single application are out of the user's hands.

However, most global colour choices are left up to the user. Perhaps the most common global use of colour is for organizing windows. Colour can be used to group related windows and to disassociate unrelated windows. There are many different approaches to window organization. For example, similar applications could have similar colours, windows for different projects could be grouped using similar colours, or windows could be coloured according to the machines that created them. What characterizes each of these organizational approaches is that in each case a different category of semantic meaning is identified by the user and windows with similar semantic meanings are grouped visually.

Colour is also excellent at helping the user resolve context. A common mistake in window systems is to accidentally type into the wrong window. However, colour has been experimentally determined to be more useful than size, shape or brightness when searching for and identifying items that vary in only one of these aspects. Additionally, colour associations are remembered longer (Meier, 1987). When two terminal windows differing only in colour are serving different functions, colour serves as an excellent way to distinguish the two.

For some people, the most exciting ability that colour provides is the ability to personalize their environment. Just as some people feel strongly about more obvious expressions of their personality, such as clothing and hair styles, so do some people want to personalize their computing environment. One user commented, for example, that "he did not care what colours were used for his windows, as long as they all were purple." Colour should also allow people to make their environment dynamic, visually rich and exciting. People often change clothing styles to reflect their mood or changes in taste. Other reflections of personal taste, such as furnishings in the home or office, or the model of car that is driven, are not changed on a regular basis only because practical considerations make this impossible. A computer window system, by its very nature, can have its appearance radically changed in an instant. Unfortunately, the tools do not exist to allow people to access this potentially exiting avenue of personal expression. Few people have the time or patience to actually create a visually interesting environment.

1.2 The Problem with Aesthetic Colour Selection

One of the reasons effective aesthetic colour selection is difficult is that it crosses many disciplines: physics, physiology, psychology, art and graphic design, to name a few. Most people who design window systems and windowing applications do not necessarily have expertise in any or all of these areas. As a result, colours are often selected seemingly at random, with no regard for even the most basic design principles. In addition, tools provided to the user for colour selection typically concentrate on allowing the user to select a single colour, ignoring how it will be used. Therefore these tools do not provide the user with any help in selecting attractive and effective combinations of colours (Meier, 1987).

With no knowledge of the various aspects of colour theory, users can do damage as well as good to their working environment, and may have no idea what is causing the problems or how to solve them. Unintentional interactions between colours can convey nonexistent meaning, such as creating illusions of window depth or unintended structural relationships. Far more seriously, insufficient colour difference can impair legibility and reading.

If colour selection represents such a hard problem, why are there no better tools available to help the user solve it? The basic answer is that colour is not well understood. There are theories explaining many aspects of

colour perception and colour theory, but no comprehensive model of human colour vision exists and explanations of some perceptual phenomena elude researchers (Boynton, 1979). Another aspect of the problem is that all of the different fields that study some aspect of colour, from graphic design to physics, use different techniques and terminology. Chapter 2 explains the aspects of some of these areas that are relevant to this thesis. Each of the many colour models mentioned in Sections 2.2 and 2.3 was created for certain reasons and is appropriate for certain applications, but most contain little information about effective colour use. More seriously, some fields give different meanings to similar terms, causing confusion.

Another reason that there are few powerful tools to aid in effective colour selection is that solving some of the problems mentioned above without unduly restricting colour choices is very hard. Creating simple tools that severely restrict the users choices is relatively easy, but tools such as this are not satisfactory. The aesthetic aspect of colour is very subjective and users do not wish to be arbitrarily restricted or to have someone elses ideas of colour harmony imposed on them. As one user interface designer put it, “A computer will never be able to tell me what colours I should like.”

1.3 Tools for Aesthetic Colour Selection

Much effort has been put into attempting to create sets of rules for colour use in computer user interfaces and window systems. Meier presents a good summary of this work (Meier, 1987). This thesis is not trying to extend or duplicate this research.

While there exist many guidelines and heuristics for effective use of colour, it is virtually impossible to implement these heuristics using current window systems. Although there have been systems which attempt to preselect colours for user interfaces, such as ACE (Meier, 1988), they are usually too specialized or too restrictive to be generally useful. In particular, there are no tools available that assist people in selecting aesthetically pleasing colours while ensuring that the basic functional colour constraints are satisfied. As a result, the application programmer or the user must consider too many different issues when selecting colours. The task is simply too complex for many users to perform effectively.

Consider the task of selecting colours for a typical window system that has four colours for each window: background and foreground colours for both the border and the interior of the window. In order to select four colours for a single window, contrast between the two background and two foreground colours must be considered. In addition, the border and window background colours must be sufficiently different that they do not blend into one another. Finally, the border background colour must be sufficiently different from the screen colour that the window stands out from the screen. The task of selecting a tasteful set of colours for this single window is not overwhelming, especially if we assume the existence of a reasonably powerful tool for selecting individual colours.

However, windows do not appear in isolation. There can be many different windows coexisting on the same display. Suppose that the person making the colour choices is aware of how colour can be used to organize these windows. For example, windows can be grouped by application, intended use, machine or some other logical relationship. With only a few applications, machines, and different uses the number of different windows can grow very quickly. For example, twenty or more different windows is not uncommon. Any of these windows may appear on the screen simultaneously, so the colours that are chosen for one window must be compatible with those chosen for all the other windows. The problem has quickly changed from one with four variables to one with perhaps eighty or more! While there are probably many solutions to this problem, they are insignificant compared to the size of the solution space. After attempting this exercise once or twice, one can see that this can be a very frustrating and quite difficult problem to solve.

What is particularly annoying for many people is that selecting colours for the interface is something that is tangential to the reason they are using the computer in the first place. As a result, most people quickly give up

attempting to discover a solution to this problem. Experience shows that users often resort to bland or extreme colour schemes in order to reduce the number of variables that they must deal with, making the task somewhat more manageable. Unfortunately, the functional benefits of colour, such as context resolution and showing logical relationships, are often the first things that are discarded in an effort to simplify the problem.

This basic premise of this thesis is that window systems can and should help the user to create and maintain an aesthetic, effective and colourful windowing environment. The aim is to demonstrate that it is possible to create a system which helps the user choose aesthetically pleasing colours without having to explicitly consider all functional colour requirements.

1.4 Previous Work

There has been relatively little work done in this area. The ACE system is the one notable exception (Meier, 1988). Other systems that automatically select colour, such as (Corte, 1986) and (Grosse, 1985), consider only a small subset of the problem and are applicable to a limited domain of colour usage. These latter two systems, for example, select sets of perceptually different colours for use in charts, tables or plots.

1.4.1 ACE: A Colour Expert System

ACE is an expert system that attempts to select colours for the sort of user interface found on the Apple Macintosh or Xerox STAR desktop environments. ACE selects its colours from a discrete subset of the HLS colour space (see Section 2.2.5), encoding relationships between all of the colours in that subset in its knowledge base. There are a few key limitations to ACE. First, the aesthetic relationships that are used to select colours are encoded in the system, with the result that the colour tastes of the authors are imposed on the user. The user can alleviate this problem only by modifying the knowledge base of the system, which is a non-trivial exercise.

Another major problem with ACE is that it was designed to select colours for the graphical elements of a single program. It selects colour using a monotonic scheme, meaning once a value has been selected for a colour it will never be changed. Therefore, the relationships between colours must be expressible as a directed, acyclic graph (DAG). While the relationships between the graphical elements of a single window may be expressible in this fashion, it is not clear that the relationships of an entire window system could be. If the relationships cannot be specified as a DAG, such a monotonic selection scheme could not be used.

More importantly, all of the relationships that ACE uses to evaluate a possible colour selection are pair-wise relationships. This precludes the possibility of expressing relationships that exist among groups of windows, for example, which is common in the sort of window systems of interest here. As Meier says, they “might be able to devise color relations between three, and maybe even four or five colors, but more than this would be extremely difficult. Adding these kinds of relations would require a fairly major overhaul ...”

1.5 The Need For a Dynamic Window System

A limitation of systems such as ACE is that they select colours for a single application without considering other windows on the display. Colours for an application’s windows need to be selected in the context of the window system as a whole, not in isolation. ACE, for example, considers multiple windows within a single application but not over multiple applications. Meier mentions that this issue should be considered, but does not attempt to solve it.

More importantly, the context within which a window's colours are selected is not static. As windows are created and destroyed, the context within which a window's colours were chosen changes and, therefore, the colours that should be used for any window may change. The colours should not be picked within a given context and then ignored when that context changes.

If this assertion seems strange, consider the analogy to window positions on the screen. At any given instant in time, there are one or more optimal spatial arrangements of the currently open windows. However, as new windows open and old windows close, this optimal arrangement changes. Thus, the windows' positions must be rearranged. And so it is with colour.

To continue with the analogy, consider the tools that are provided by window systems to help the user organize the positioning of their windows. The most common tool is a simple interactive technique which allows the user to tell the system exactly where to move a window. More powerful tools than this are available, however. Some systems will select an initial window position if one is not given, effectively allowing the user to say "I don't care where it goes on the screen." If the position is unacceptable, the user can easily change it. The key point is that the user does not have to specify a position unless the one chosen by the system is unacceptable. Thus the user has one less specification to make.

There are other examples of powerful tools for positioning windows. Tiled window systems are based on the assumption that opened windows should not overlap, and automatically rearrange the windows to enforce this. Whether this is desirable behaviour is a matter of personal taste. A more powerful system is Stack Windows (Png, 1991), which allows windows to be hierarchically grouped into common areas called *stacks*. Stacks are windows which contain other windows and have tiling behaviour enforced within them. Repositioning of windows within the stack is automatic when windows are opened or closed. Tools are provided to manipulate the stacks and to override the arrangement of windows within the stacks. Any window or stack that is not contained in another stack can be overlapped freely; thus, the power of a tiled window system is available within the stacks and the freedom of non-tiled window systems to overlap windows is available outside the stacks.

The most powerful tool implemented to date is the *jostling window system* used in Schlueter (1990) to implement the concept of *perceptual synchronization*. Perceptual synchronization is based on the belief that the screen is a resource that is shared by the user and the applications to communicate. Different applications make different demands on the screen space that may cause perceptual problems on the part of the user. These problems, such as text or graphics appearing to flow between windows, should be detected and fixed. This process is referred to as synchronizing the users' perception of the system. While the actual test system had usability problems, none were of the sort that could not be fixed in a production system. The system could be thought of as attaching a higher level of semantic meaning to window positioning, and using these semantics to help the user position the windows.

Unfortunately, the tools provided by window systems for manipulating colour are surprisingly primitive. No window system provides anything more than the most basic support for colour selection, forcing the user to specify every colour explicitly. Other problems can be highlighted by returning to the analogy of window positioning. Many window systems do not provide a general facility for users to modify the colours of a window after the window has been created. Thus, window colours can be specified only when a window is first created unless the client program specifically contains functions for changing its colours. This is analogous to specifying an initial position for a window and not being able to move the window unless the application provides a facility which allows it to be repositioned! Obviously, the resource dictated the availability of the tools: screen real estate is quite limited compared to the size of the colour space, imposing a more immediate demand on providing the capability to reposition windows.

Other window systems, such as that found on the Macintosh, allow the user to change the colour of existing windows but the tools for colour selection are still not as powerful as those for window positioning. In the case of the Macintosh, for example, windows are positioned for the user the first time they are created in a random

fashion, spreading them out over the screen. Colours, on the other hand, are always the same by default, providing not even the slightest variation that is not specified by the user.

Given the situation mentioned above, why are there not good colour tools for current window systems? The key element that is missing from most existing window systems which is that there is no semantic meaning attached to the colours on the display, making it impossible for the window system to provide any sort of reasonable help to the user. By attaching semantic meaning to colours, the window system should be able to assist the user in solving the problem of effective and aesthetic colour selection.

1.6 Goals

The kind of tool created here is something analogous to the jostling window system (Schlueter, 1990). This system attached semantic meaning to windows and their contents and defined relationships between these elements of the display. The system imposed constraints on the possible positions of the windows based on these relationships. Most importantly, the system dynamically adjusted the arrangement of the windows to satisfy these constraints.

Similarly, by attaching semantic meaning to colours, it is possible to define relationships between these colours and to impose constraints based on these relationships. Using these constraints, it is possible to create a window system which will assist users in making aesthetic colour choices by both making some of the choices for them and removing some of the constraints on the choices they need to make. For example, the following abilities are provided:

- Functional constraints are be handled automatically, allowing the user to ignore them if they so desire.
- Many users have no training in art or graphic design. Therefore, the system will suggest possible aesthetic colour combinations to the user.
- The user is able to make abstract colour specifications and have the system infer the details.
- The system provides reasonable defaults.
- The system can be gradually customized, allowing a user to specify as little or as much as he or she wants.

However, it should be kept in mind that the overall goal of this thesis is to create a system that demonstrates that more powerful tools for assisting with aesthetic colour selection are possible, not to provide definitive solutions to all of the issues associated with the abilities mentioned above. The satisfactory solutions to many of the problems discussed in this thesis represent theses on their own.

In Chapter 2, the reader will be provided with the background in colour theory necessary to understand and appreciate the bulk of this thesis. Chapter 3 will describe one view of colour use that will prove particularly helpful in creating this system. A metric for calculating contrast will be devised in Chapter 4. Chapter 5 will develop the the colour constraints that are needed to create a dynamic window system to assist with colour selection. The implementation of a prototype version of the dynamic window system is described in Chapter 6. Finally, Chapter 7 discusses the conclusions that can be drawn and future work that needs to be done.

Chapter 2

Background

In order to understand the body of this thesis, a variety of background information is required. In the following sections several topics are discussed in sufficient depth to understand the thesis. For each topic, references are provided for further reading. Knowledge of computer science and a basic knowledge of computer graphics are assumed. For an introduction to computer graphics, see (Foley et al., 1990).

The purpose of this thesis is to demonstrate that a dynamic window system is a feasible approach to assisting users with the selection and maintenance of window colours. Colour science bridges many disciplines, including computer science, physics, physiology, psychology, art and graphic design. In order to create powerful colour tools, selected aspects of each of these subjects must be understood. The logical starting point is video display hardware. Given an understanding of how coloured images are created, the underlying colour models that are used throughout this thesis for specifying colour values are explained. Following this, psychological aspects of colour are examined, including a brief look at colour perception, the effects of colour on reading, and colour discrimination. Finally, artistic models of colour and colour harmony are investigated to discover techniques of assisting with harmonious colour selection.

2.1 Colour Display: The Colour CRT

Currently, the most common video display hardware is the *cathode ray tube* (CRT), an electronic device that produces patterns of light on a glass display. A cathode creates electrons which are accelerated towards the front of the CRT. This stream of electrons is focused into an electron beam which converges to a small point at the faceplate of the CRT. This beam is aimed at the phosphor-coated inside of the display surface. When the electrons strike the phosphor, it emits visible light. The pattern of light created by the glowing phosphor is the image seen on the display surface. To create colour, several phosphors are needed, and the excitation of each must be controlled independently. The mechanism for doing this is the *shadow mask*. The inside of the tube's viewing surface is covered with phosphor dots arranged in triangular patterns, as shown in Figure 2.1. These phosphor dots are extremely small, a dozen or more being required to create a single *pixel* (or point on the screen.) The colour of these phosphors are red, green and blue. Three different electron guns, arranged in the same triangular shape as the phosphors, are independently focused at the same point on the screen. The shadow-mask is placed between the phosphor surface and the incoming electrons, and has small holes in it, one for each set of three phosphors. The electron guns, shadow-mask holes and phosphors are precisely arranged so that only the electrons from one gun strike any individual phosphor, as shown in Figure 2.2. This arrangement allows one gun to control the intensity of each of the red, green and blue phosphors. All colours that are displayed on a computer screen are

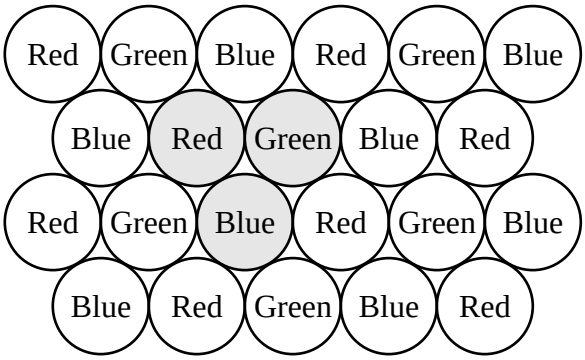


Figure 2.1: The Triangular Pattern of Red, Green and Blue Phosphors.

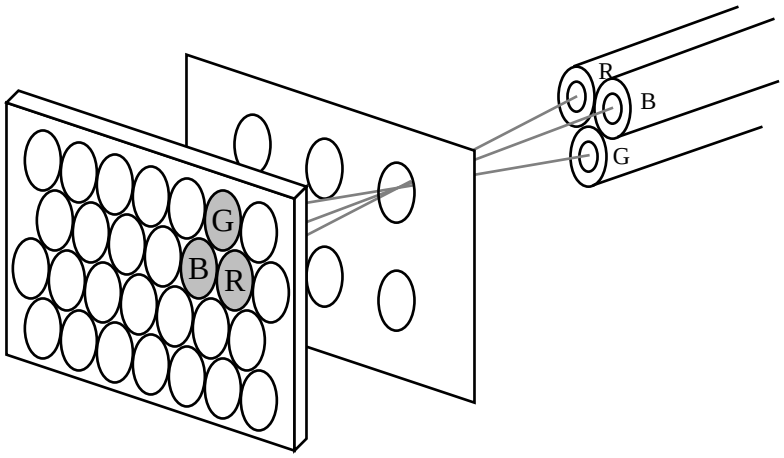


Figure 2.2: The Shadow-mask CRT.

created as a linear combination of spectral power distributions of the red, green and blue phosphors, as discussed in Section 2.2.1. For example, if each of the phosphors could be illuminated to two different levels (on and off) there would be eight possible combinations. More commonly, the phosphors can be illuminated at 256 different intensities, giving over sixteen million possible combinations. A more in-depth description of the workings of a CRT, as well as other display devices, may be found is books such as (Foley et al., 1990). For simplicity, the CRT stands for all video display devices, such as plasma and LCD displays, because their capabilities are similar.

2.1.1 Gamma Correction

Controlled display of many different intensities on a CRT is not as straightforward as it seems, since the intensity of the light output by a phosphor is not linearly related to the number of electrons striking it. For simplicity, the internal workings of the CRT are ignored and only the relationships between the voltage and the light output for a given phosphor are considered[1]. The relationship is

$$I = kV^{\gamma}, \text{ or } V = (I/k)^{1/\gamma} \tag{2.1}$$

for constants k and γ. (Foley et al., 1990) describes how to actually calculate the value of required voltage V for a desired intensity I. For a typical CRT, the value of γ falls in the range 2.2 to 2.5. What is important here is to note that the relationship is not linear and that the values of k and γ vary between CRTs. Correcting for the non-linearity of phosphor response in this manner is referred to as *gamma correction* because of the historical use of gamma as the exponent in Equation 2.1.

Unfortunately, the situation is even more complicated. There are two other factors which contribute to the perceived response of the phosphors:

- Many current CRT's are equipped with dials that control the "brightness" or "black level" and the "contrast" of the display. The brightness control dial, in particular, changes the response of the relationship between the voltage received by the monitor and the number of electrons striking the phosphor surface, and thus the required gamma value. Experience has shown that the gamma value for a CRT with a typical brightness control can fall in a range of 1 to 5!
- The lighting conditions in the area where the CRT is being viewed dramatically affects the perceived intensity of the phosphors (see the discussion of *veiling lights* in Section 2.5.)

These two factors are, in fact, related — the existence of the second necessitates the first. Brightness controls are put on monitors to allow users to compensate for lighting conditions in the area of the CRT.

2.2 Colour Models

Colour is intuitively described using the three terms *hue, lightness* and *saturation*. Hue refers to the chromatic component of the colour, the quantity that distinguishes between colours such as red, green and blue. Lightness refers to how dark or light a colour is. Finally, saturation describes the purity of a colour, which ranges from neutral grey to pure colour. The colours black, white and the shades of grey are said to be "achromatic" or neutral — they are completely desaturated, having no "chromatic" colour component.

For colour to be used effectively in computer graphics, it is necessary to be able to specify colours precisely. Most computer framebuffer hardware require colour values to be specified by their red, green and blue (RGB)

[1] The relationship between the number of electrons striking the phosphor and the voltage sent to the CRT for that phosphor is linear.

components. More abstract models of colour specification, such as HSV and HLS, are also used in computer graphics. These models provide a more intuitive organization of colour values, but are merely different views of the same RGB colour space. While these models are precise with respect to the values placed in the video framebuffer, they do not allow accurate colour specification with respect to colour appearance — the same RGB, HLS or HSV specification may appear as a different colour on different computers or different CRTs. Objective colour specification, independent of any display medium, is provided by the branch of physics known as *colorimetry*. Many of these models are based on colour models developed long ago by artists and scientists such as (Chevreul, 1967), (Munsell, 1947) and (Ostwald, 1931).

In the following sections, several models will be described. For a more complete history of colour models, see (Norman, 1990) or (Gerritsen, 1988). For a more in depth discussion of the colour models described below, see one of (Foley et al., 1990; Rogers, 1985; Hall, 1988; Boynton, 1979). In addition, many of the concepts discussed in this section are treated at an introductory level in (Hope and Walch, 1990).

2.2.1 Colour Mixing

From the psychophysical point of view, colour can be defined as (Wyszecki and Stiles, 1982)

> that characteristic of a visible [colour stimulus] . . . by which an observer may distinguish differences between two structure-free fields of view of the same size and shape, such as may be caused by differences in the spectral composition of the radiant power concerned in the observation.

This *spectral power distribution* which defines a colour is a measurement of the intensity of light at unit intervals for each visible wavelength of the electromagnetic spectrum, which extends from approximately 400 (λ_{min}) to 700 (λ_{max}) nanometers, and is denoted as follows:

$$E(\lambda) = \sum_{i=\lambda_{min}}^{\lambda_{max}} e_i \tag{2.2}$$

where e_i is the intensity of wavelength λ_i. When all of the visible wavelengths are present in approximately equal amounts, the result is achromatic. Otherwise, a *dominant wavelength* is perceived which corresponds to the intuitive notion of hue.

When colours are mixed, their spectral power distributions are mixed. There are two colour mixing systems, *subtractive* and *additive*, which apply to reflective and self-luminous objects, respectively.

Additive When working with luminous objects (objects that generate light), the colour primaries typically used red, green and blue. Luminous objects create colour by emitting light with certain spectral power distributions. Colour mixing is linearly additive. When two lights with spectral power distributions E_1 and E_2 are mixed together, the result E_3 is

$$e_{3i} = e_{1i} + e_{2i}, \qquad i = \lambda_{min}, \ldots, \lambda_{max} \tag{2.3}$$

where $e_{j\,i}$ is the component of E_j at the wavelength λ_i. When the primaries are added together in equal quantities, all the wavelengths of light are present and the result is seen as white light[2]. These three colours can be combined to create all of the colours of the spectrum plus purple (a non-spectral colour with is created by adding together red and blue). This allows a wide variety of colours to be created. Anyone who has watched TV or used a computer has seen colour created additively.

[2]This is a simplification. See Section 2.2.2.

Subtractive When most people think of the *primary colours* (the basic colours that are combined to create all other colours) their intuition is based on their childhood experience of mixing paint: yellow and blue combine to produce green. The three primaries are *blue*, *red* and *yellow*. These colours are the primary colours for a *subtractive* colour scheme[3], which is the colour scheme used when mixing paint, ink, dye, or any other reflective material, and when dealing with transmissive materials such as film. Subtractive colour mixing is significantly more complicated than additive mixing, so a detailed account of how it occurs is not possible here. A brief description of how subtractive mixing occurs with filters is all that is possible.

Filters do not generate light, but are seen by the light which passes though them. The color of a filtered light is, therefore, determined by the amount of each wavelength that passes though the filter. This can be expressed by a transmission function

$$T(\lambda) = t_i, \qquad 0 \leq t_i \leq 1, i = \lambda_{min}, \ldots, \lambda_{max} \tag{2.4}$$

where t_i is the fraction of wavelength λ_i that is transmitted. When a light with a spectral power distribution $E(\lambda)$ is passed through a filter with a transmission function $T_1(\lambda)$, the spectral power distribution $E_1(\lambda)$ of the resulting light is

$$e_{1i} = e_i t_{1i}, \qquad i = \lambda_{min}, \ldots, \lambda_{max} \tag{2.5}$$

If this light is then passed through a second filter, with transmission $T_2(\lambda)$, the spectral power distribution $E_2(\lambda)$ of the resulting light is

$$e_{2i} = e_i t_{1i} t_{2i}, \qquad i = \lambda_{min}, \ldots, \lambda_{max} \tag{2.6}$$

In the above situations, where light is transmitted through one or more filters, the effect is to subtract intensity from the spectral power distribution of the incident light. That is why this is referred to as subtractive colour mixing — mixing coloured objects results in light being subtracted from the incident light. When the three filters corresponding to ideal subtractive primaries are used in succession, little colour is transmitted and the result is black.

The question of how pigments combine requires knowledge of how pigments are suspended in the medium and how light is scattered within the medium. For a more detailed discussion of this, see (Cowan and Ware, 1985).

Complementary Colours With both additive and subtractive colour mixing, each pure colour has associated with it another colour, call its *complement*. Mixing a colour with an equal amount of its complement results in grey. Mixing a colour with a lesser amount of its complement results in a less saturated colour of the same hue. The complements of the primary colours are often referred to as the *secondary* colours.

2.2.2 Colorimetry

The purpose of *colorimetry* is to provide an objective, quantitative way of specifying colour. Standard colorimetric systems, which are based on additive colour mixing, are well adapted for use with CRTs.

2.2.2.1 Colour Matching

Colorimetry begins with colour matching. Imagine an experiment where the subject is looking at two areas of coloured light. One area is a test colour. The subject attempts to create a matching colour in the other area. To perform this match, the subject has three dials which control the intensity of three coloured lights (referred to as the *colour primaries*) that are combined to create the matching colour. These three lights are red, green and

[3]The subtractive primaries are actually *cyan*, *magenta* and *yellow*, but artists typically refer to the primaries as red, blue and yellow.

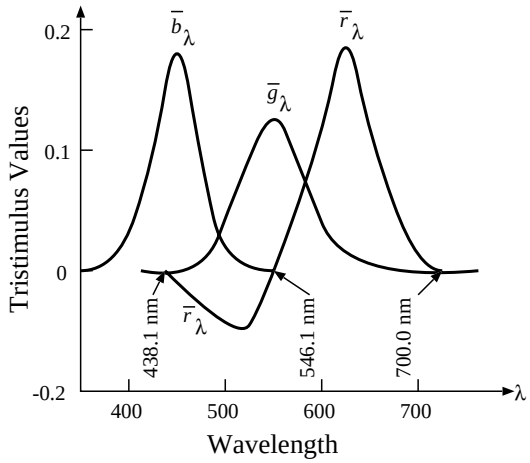


Figure 2.3: Colour-matching functions, showing the tristimulus values for the visible spectrum, where the three primaries are R = 700nm, G = 546.1nm and B = 438.1nm. (Adapted from Foley et al. (1990))

blue, the additive primaries from section 2.2.1. In practice, some colours can not be matched by the positive combinations of these standard lights. This problem can be overcome by allowing the subject to add one of the primaries to the test light and then attempt to create a match with the remaining two primaries. In this case, the match can be specified as a combination of positive and negative amounts of the primary lights, where a negative amount means the primary was added to the test light instead of the match light. These values are referred to as *tristimulus values* for the colour, since they define the amount of each stimulus needed to create the colour.

By performing this experiment repeatedly, with the test light set successively to every visible wavelength of light, the amount of each primary needed to match any wavelength of light can be tabulated. This was done with a set of observers and the results averaged to obtain values for a *standard observer*. This table defines three colour-matching functions, called $\bar{r}_\lambda$, $\bar{g}_\lambda$ and $\bar{b}_\lambda$, similar to those shown in Figure 2.3 (which are adjusted so that the area under each of the three functions is 1.0). In order to obtain a colorimetric specification for a stimulus light with spectral power distribution E_λ the tristimulus values for the light can be calculated using the equations

$$\begin{aligned} R &= \int E_\lambda \bar{r}_\lambda d\lambda \\ G &= \int E_\lambda \bar{g}_\lambda d\lambda \\ B &= \int E_\lambda \bar{b}_\lambda d\lambda \end{aligned} \tag{2.7}$$

An important point to note is that two colours with physically different spectral distributions can result in the same tristimulus values. These colours will appear the same for the standard observer, and are referred to as being *metameric*. The pairs of colours are called *metamers*.

2.2.2.2 The CIE System

The above colour matching experiment was performed and the data used to create an objective description of colour. This work was carried out by the Commission Internationale de l'Éclairage (CIE), which in 1931 defined

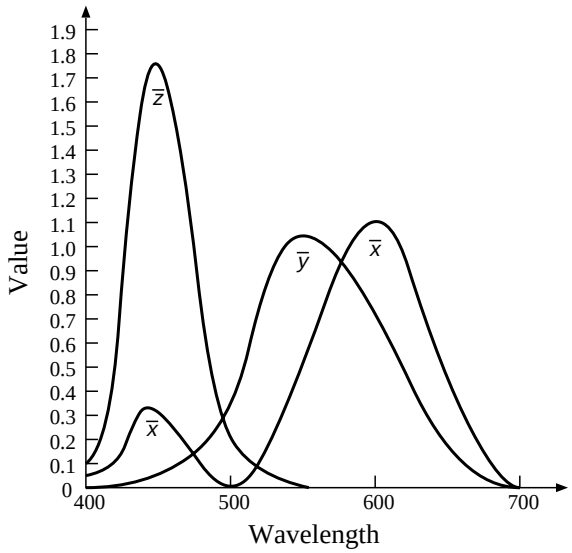


Figure 2.4: Colour-matching functions, showing the tristimulus values for the CIE 1931 primaries X, Y and Z. (Adapted from Foley et al. (1990))

the CIEXYZ colour space, which uses three hypothetical primaries with colour-matching functions $\bar{x}(\lambda)$, $\bar{y}(\lambda)$ and $\bar{z}(\lambda)$ shown in Figure 2.4. These standard primaries were specifically chosen to satisfy a few properties, most importantly that all visible colours have positive tristimulus values, and that the *luminous energy* of the colour is represented by the Y value. To satisfy these constraints, the primaries themselves are imaginary and can not be created from non-negative spectral power distributions. Analogous to Equation 2.8, the tristimulus values for a stimulus light E_λ can be calculated as follows

$$\begin{aligned} X &= \int E_\lambda \bar{x} d\lambda \\ Y &= \int E_\lambda \bar{y} d\lambda \\ Z &= \int E_\lambda \bar{z} d\lambda \end{aligned} \tag{2.8}$$

The visible colours create a cone-shaped volume in XYZ space, with black at the origin, as show in Figure 2.5.

While the Y value, which defines the luminance of the colour, corresponds roughly to the intuitive notion of brightness, the XYZ tristimulus values do not correlate neatly with the intuitive notions of hue and saturation. To separate the chromatic component from the brightness, the CIE defined the *chromaticity* coordinates of a colour (which define the hue and saturation, but ignore the luminance) as

$$\begin{aligned} x &= \frac{X}{X+Y+Z} \\ y &= \frac{Y}{X+Y+Z} \\ z &= \frac{Z}{X+Y+Z} \end{aligned} \tag{2.9}$$

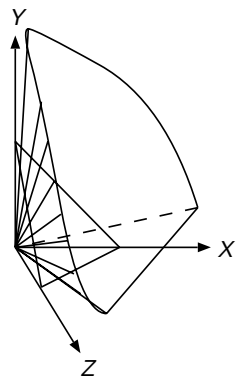


Figure 2.5: The visible colours in XYZ space form a cone radiating up into the positive octant. The plane $X + Y + Z = 1$ is also shown (Adapted from Foley et al. (1990))

Notice that $x + y + z = 1$, so only two values, x and y, need to be retained, because $z = 1 - x - y$. However, in order to recover X, Y and Z, a third piece of information is needed. That value is the luminance, Y. The values x and y can be plotted on the CIE chromaticity diagram, shown in Figure 2.6, which represents the plane $X + Y + Z = 1$. These three values make up what is called the CIE Y, x, y system. Notice on the diagram that the pure wavelengths of light are plotted around the horseshoe shaped periphery. Colours then gradually blend until they reach white near the center. The CIE also defined a number of "standard" illuminants, which approximate the spectrum of light produced by various white light sources. *Standard illuminant* D_{65}, which represents sunlight, is shown on the diagram.

An advantage of the CIE chromaticity diagram is that colour mixing is a linear function. In other words, any colour produced by adding together positive amounts (not negative amounts, as was allowed in the colour matching experiment) of colours I and J falls on the line IJ that connects them. Extending this, any additive combination of three colours I, J and K lies in the triangle IJK. This area, called a *colour gamut*, represents all the colours that can be additively produced using I, J and K as colour primaries (Foley et al., 1990). Colour gamuts are actually three dimensional since the verticies are defined using XYZ coordinates, but are more conveniently represented on the two-dimensional chromaticity diagram.

One problem with the CIE XYZ system is that it is not *perceptually uniform*, meaning the Euclidean distance between any two points in XYZ space does not indicate the size of the perceived difference between the two colours. There have been many colour spaces developed which attempt to represent colour in a perceptually uniform fashion. All these colour spaces are non-linear transformations from the XYZ colour space. One, the OSA Uniform Colour Space, is discussed in Section 2.3. Another, the $L^*u^*v^*$ colour space, has been standardized by the CIE. Given the coordinates of the white colour as (X_0, Y_0, Z_0), the transformation of any colour from XYZ to $L^*u^*v^*$ is defined by

$$\begin{aligned} L^* &= 25\left(\frac{100Y}{Y_0}\right)^{1/3} - 16; \qquad (1 \le Y \le 100) \\ u^* &= 13L^*(u' - u'_0) \\ v^* &= 13L^*(v' - v'_0) \end{aligned} \tag{2.10}$$

where

$$\begin{aligned} u' &= 4X/(X + 15Y + 3Z), \qquad v' = 9Y/(X + 15Y + 3Z) \\ u'_0 &= 4X_0/(X_0 + 15Y_0 + 3Z_0), \quad v'_0 = 9Y_0/(X_0 + 15Y_0 + 3Z_0) \end{aligned} \tag{2.11}$$

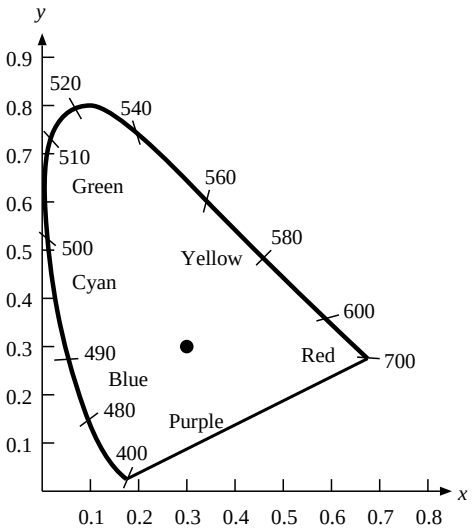


Figure 2.6: The CIE 1931 Chromaticity Diagram (Adapted from Foley et al. (1990))

$L^*u^*v^*$ (also called CIELUV) is important because measurements of colour difference must be performed in a uniform colour space. Of all the uniform colours spaces, CIELUV is computationally the easiest to transform colour coordinates into.

2.2.3 The RGB Colour Model

For computer applications, the dominant colour model is the RGB model. This model specifies colours as a additive combination of the red, green and blue primaries of the display. It is in widespread use owing to its close correspondence to colour CRT hardware.

The RGB colour model can be visualized as a cube of unit size (each side has a length of 1 unit.) Its three orthogonal axes are the intensities of the red, green and blue primaries. These primaries, which are produced by the three coloured phosphors used by the CRT display, mix additively. When all primaries are zero, the phosphors are off and the resulting colour is black. Conversely, when all primaries are 1, each phosphor provides its maximum contribution and the resulting colour is white. When all phosphors contribute the same amount, the result is a shade of grey. Therefore, achromatic colours appear along the main diagonal of the RGB cube, between black (0,0,0) and white (1,1,1) (Figure 2.7.)

The gamut of available colours on a CRT is determined by the chromaticity coordinates of the CRT phosphors. Two monitors with different phosphors have different gamuts and display the colours of the RGB cube differently. For this reason, it is often desirable to specify colour using a display independent model such as the XYZ model.

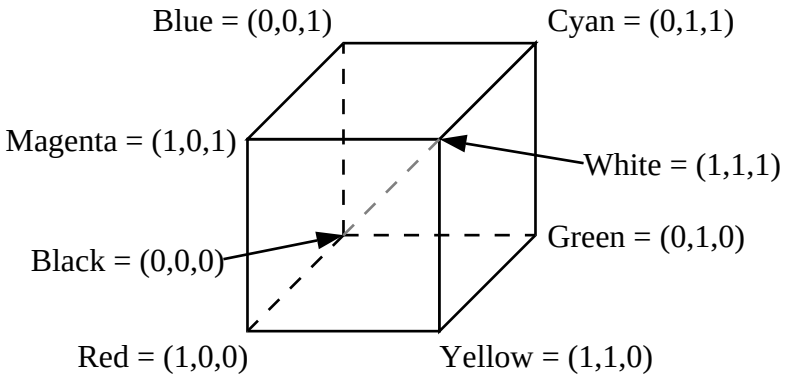


Figure 2.7: The RGB Colour Model

The conversion between RGB and XYZ can be expressed as follows (Hall, 1988, Eq 3.2)

$$\begin{array}{lllll} \text{red phosphor:} & r_x, & r_y, & r_z = 1 - r_x - r_y \\ \text{green phosphor:} & g_x, & g_y, & g_z = 1 - g_x - g_y \\ \text{blue phosphor:} & b_x, & b_y, & b_z = 1 - b_x - b_y \\ \text{white point:} & w_x, & w_y, & w_z = 1 - w_x - w_y \end{array} \tag{2.12}$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} S_r(r_x) & S_g(g_x) & S_b(b_x) \\ S_r(r_y) & S_g(g_y) & S_b(b_y) \\ S_r(r_z) & S_g(g_z) & S_b(b_z) \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

S_r, S_g and S_b are scale factors for the red, green and blue. For more information on generating this matrix, see (Hall, 1988). It should be noted that, like the XYZ space, the RGB cube is not perceptually uniform.

2.2.4 The HSV Colour Model

In contrast to the hardware oriented RGB colour model, the HSV model attempts to provide a model based on the intuitive notions of **H**ue, **S**aturation and lightness (**V**alue.) The HSV model is also known as the HSB model, where B stands for **B**rightness. The model defines an inverted single-hexcone (Figure 2.8) with a height and radius of 1. Hue is measured angularly around the vertical axis, with red at 0°. Value ranges from 0 at the tip of the cone to 1 at the flat top. Saturation is a ratio of the distance from the center vertical axis to the triangular side, ranging from 0 to 1. The additive primaries and their complements are spaced equally around the edge of the flat top of the cone, at 0°, 60°, 120°, 180°, 240° and 300°, with the primary and its complement opposite each other (180° apart.) They are the maximally saturated colours, having $S = 1$ and $B = 1$.

The HSV model is actually a deformation of the RGB model. Looking down the principle axis of RGB cube gives an intuitive notion of how the two correspond, as shown in Figure 2.9. Because this model is a transformation of the RGB model, it shares the same gamut problems. For example, a colour with a red hue (0°), .5 saturation and .75 lightness are not colorimetrically the same on different monitors.

The conversion from RGB to HSV is

$$M = \max(r, g, b)$$

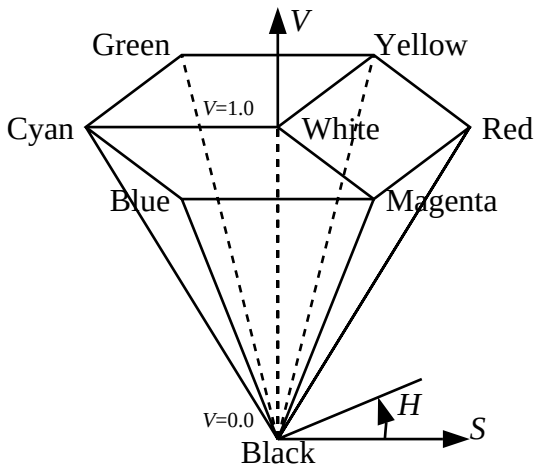


Figure 2.8: The HSV Colour Model

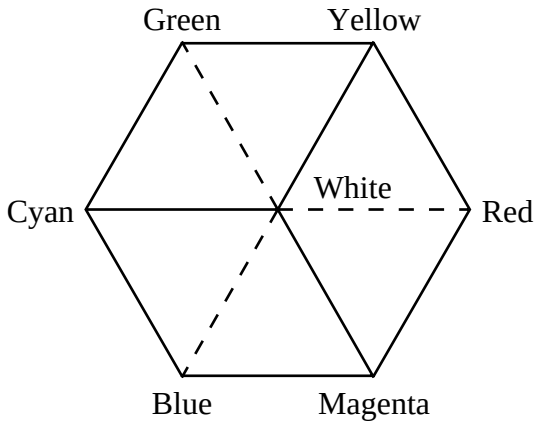


Figure 2.9: The RGB cube, when viewed along the principle axis, with white at the front and black hidden at the back, presents the same view as the top of the HSV hexcone.

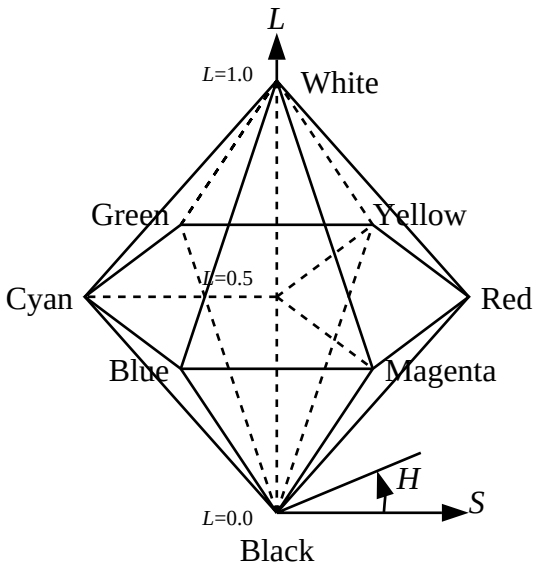


Figure 2.10: The HLS Colour Model

$$
\begin{aligned}
m &= \min(r, g, b) \\
V &= M \\
S &= \begin{cases} 0 & \text{if } M = 0 \\ (M - m)/M & \text{otherwise} \end{cases} \\
H &= \begin{cases} \text{UNDEFINED} & \text{if } S = 0 \\ 60((g - b)/M - m) & \text{if } r = M \\ 60(2 + (b - r)/M - m) & \text{if } g = M \\ 60(4 + (r - g)/M - m) & \text{if } b = M \end{cases}
\end{aligned} \tag{2.13}
$$

For more information on the HSV colour model, including detailed algorithms for conversion between RGB and HSV, see (Foley et al., 1990; Rogers, 1985).

2.2.5 The HLS Colour Model

Another model that attempts to provide an intuitive method of colour specification is the **H**ue, **L**ightness and **S**aturation (HLS) model. The HLS model can be pictured as a double hexcone, as shown in Figure 2.10. Like the HSV model, HLS is a transformation of RGB. Given the discussion of the HSV model above, however, it may be easier to consider the HLS model as a transformation of HSV. Visually, the transformation is simple. In HSV, the black point is at the tip of the single hexcone and the white point is in the center of the flat top. In HLS, the black point is at one tip of the double hexcone and the white point is at the other. This results in the ring of saturated colours, which is at the flat top of the single hexcone in HSV, being half way up the lightness axis where the two single hexcones meet. Numerically, this means that a lightness of 0.5 is required to obtain the maximally saturated colours, as opposed to a value of 1.0 in HSV.

HLS offers a slightly more intuitive method of colour selection than HSV. With the fully saturated colours in the middle of the lightness axis, the lighter shades (or *tints*, as they are sometimes called) of a colour are all in the upper hexcone, with white at the top. Conversely, the darker shades (or simply *shades*) of a colour are all in the lower hexcone, with black at the base. This corresponds more naturally to the way an artist mixes colour, as will be discussed in Section 2.4. By visualizing the L=0.5 plane as being the *tones* obtained by mixing together pure colours, the upper hexcone is, by analogy, the result of adding white to any of the tones. Similarly, the lower hexcone is the result of mixing black with the tones.

The conversion from RGB to HLS is very similar to Equation 2.13, the important difference being the calculation of the L coordinate compared to that of the V coordinate of HSL. The conversion is

$$\begin{aligned} M &= \max(r,g,b) \\ m &= \min(r,g,b) \\ L &= (M-m)/2 \\ S &= \begin{cases} 0 & \text{if } M = m \\ (M-m)/(M+m) & \text{if } L \leq 0.5 \\ (M-m)/(2-M-m) & \text{otherwise} \end{cases} \\ H &= \begin{cases} \text{UNDEFINED} & \text{if } M = m \\ 60((g-b)/M-m) & \text{if } r = M \\ 60(2+(b-r)/M-m) & \text{if } g = M \\ 60(4+(r-g)/M-m) & \text{if } b = M \end{cases} \end{aligned} \tag{2.14}$$

For further information on the HLS model, including detailed algorithms for conversion between RGB and HLS, see (Foley et al., 1990; Rogers, 1985).

2.2.5.1 Value, Lightness and Brightness

While providing more intuitive approaches to colour selection than the RGB model, the HSV and HLS models do not solve the problem of perceptual non-uniformity. Furthermore, there is a new problem that must be kept in mind, which is that the neither value nor lightness corresponds to the intuitive notions of lightness or brightness.

Technically, *brightness* is the intensity of a light source or other self-luminous object, such as a CRT. *Lightness* is the perceived intensity of a reflective object. *Value* is a carefully calibrated uniform lightness scale used in the Munsell colour system (Munsell, 1947).

While these terms are used in colour models such as HLS, HSV or HSB, the reader should be aware that they are used in these systems to represent fairly arbitrary measures of colour luminance and should not be confused with the technical meanings of these terms.

For example, analysis of the fully saturated colours (HSV value $= 1$, HLS lightness $= 0.5$), shows that they do not all have the same perceived brightness. Table 2.1 shows the XYZ tristimulus values and the chromaticity coordinates, when viewed on a typical colour CRT[4], of the three RGB primaries and their complements. A glance at the Y values (which is proportional to the perceived brightness of the colours) shows that they are far from uniformly bright, yet they all have a value of 1.

[4]The chromaticity coordinates of the monitor primaries and white point are those of a typical colour CRT, as suggested in (Hall, 1988).

Colour	RGB			XYZ			Chromaticity	
White	1	1	1	0.951	1.000	1.088	0.313	0.329
Red	1	0	0	0.589	0.290	0.000	0.670	0.330
Green	0	1	0	0.179	0.605	0.068	0.210	0.710
Blue	0	0	1	0.183	0.105	1.020	0.140	0.080
Cyan	0	1	1	0.362	0.710	1.088	0.168	0.329
Magenta	1	0	1	0.772	0.395	1.020	0.363	0.181
Yellow	1	1	0	0.768	0.895	0.068	0.444	0.517

Table 2.1: The XYZ tristimulus values and the chromaticity coordinates for a standard set of RGB primaries and their complements.

2.2.6 The Gerritsen Model

While the HSV and HLS models are an improvement over the RGB model with respect to intuitive colour selection, they suffer from the problem that the perceived brightness of the colour does not correspond to the lightness/value parameter of the model. This is to be expected because the actual colours created by models depend on the characteristics of the CRT. Without knowing the chromaticity coordinates of the CRT phosphors the perceived brightness of any colour can not be accurately predicted. Furthermore, even when the chromaticity coordinates are known, it may not be possible to predict the perceived brightness confidently because of other factors that influence colour appearance (see Section 2.5.)

However, even if it is impossible to discover the exact perceived brightness of a colour, it is possible to improve on than the HLS and HSV models. Consider the Y values listed in Table 2.1. They range from .1 to .9, whereas the value parameter for the HSV model is always 1.0 and the lightness parameter for the HLS model is always 0.5. Furthermore, the variation is due to the nature of both the phosphors used and the human eye. The values for red, green and blue in this table correspond to the phosphors and the relationship between them, that of green being the brightest, red being darker and blue being the darkest. This general relationship is a function of the human visual system and, therefore, holds for all CRTs, even if the exact numbers vary.

Gerritson noted this and proposed a new model which he refers to as the *Colour-Perception-Space* (Gerritsen, 1975; Gerritsen, 1988) shown in Figure 2.11. His model corresponds to the HLS space with the important difference that the fully saturated colours do not all lie on the $L = .5$ plane. Instead, he codes the Lightness scale in nine intervals and labels them A-J, where white is A, black is J and the middle grey falls between E and F. The six primary and secondary colours are arranged around the outside of the hue ring identically to the HLS space, but that ring is now deformed so that the fully saturated colours have a Lightness corresponding (roughly) to their luminance. Gerritsen ignores the problem of colours having a different appearance on different CRT's, and sets the colours to values that are indicative of what he believes to be their correct perceptual relationship to each other. Table 2.2 shows the Lightness values Gerritsen uses compared to the luminances of the hypothetical monitor in Table 2.1. In this case, while Gerritsen's values have the same relationship to each other as the CRT, the green that Gerritsen uses is much darker than the green phosphor of the hypothetical CRT. As a result, the green value is darker than its complement magenta, whereas our hypothetical CRT has a green that is brighter than its complement magenta. However, perceived brightness is not exactly equivalent to luminance, so his values are actually more reasonable than suggested by Table 2.2. In any event, the Lightness values of the model more closely reflect the perceived brightness of the colours than the HLS or HSV models.

The idea behind Gerritsen's model, regardless of its shortcomings, is a good one. It is often desirable to specify something akin to the perceived brightness of a colour when making colour selections, such as when two or more

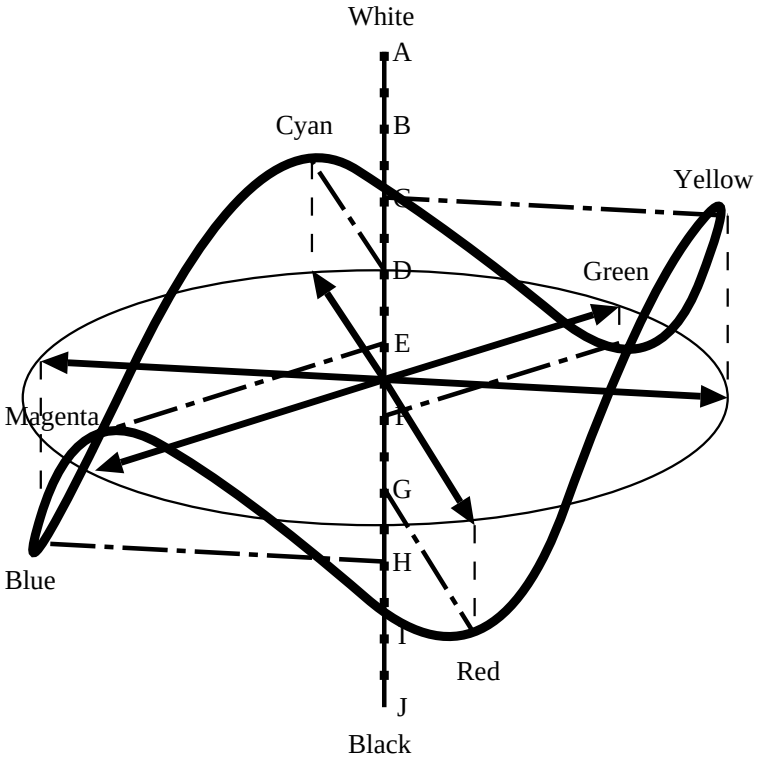


Figure 2.11: Gerritsen's Colour-Perception-Space (Adapted from Gerritsen (1988))

Colour	Gerritsen	Y
Red	0.33	0.29
Green	0.44	0.61
Blue	0.22	0.10
Cyan	0.66	0.71
Magenta	0.55	0.39
Yellow	0.77	0.90

Table 2.2: Gerritsen Lightness Compared to Colour Luminance on One CRT

colours of the same perceived brightness are needed. Gerritsen's Colour-Perception Space allows this, whereas the RGB, HSV and HLS models do not except via experimentation.

2.3 The OSA Colour Space

Another useful uniform colour space was created by the Optical Society of America (OSA) Committee on Uniform Color Scales. It consists of a set of 424 painted samples. These tiles are arranged as a lattice of points in a regular-rhombohedral crystal in a Euclidean colour space. The colour space is perceptually uniform, in that equal distances between points correspond to equal visual differences between the corresponding colours. The organization of the space was chosen so that each point has 12 equally spaced neighbours and would appear in six entirely different equally spaced uniform scales. This organization is based on Wyszecki's *uniform colour space*, which is in turn derived from the Munsell space (Wyszecki, 1954), and is chosen so that

> the system of colour sampling ... provides the maximum number and variety of [one and two dimensional] uniform colour scales and exhibits the maximum possible variety of relationships among colours (MacAdam, 1974).

The colour difference information for the space was experimentally derived by having subjects judge pairs of 59 sample tiles for colour difference. The data for the subjects was analyzed and a mathematical model was derived that best fit this data. The final 424 points in the colour space were generated from this mathematical model and checked further.

A shortcoming of the OSA space is that the equations used to derive the colours do not extrapolate well outside the range of the original colour tiles. The final set of tiles encompases a subset of the possible colours that are available on a CRT, for example, and the equations become very badly behaved within the gamut of a typical CRT. Work has been done to try and create a model that is more comprehensive compared to the CRT gamut (Seim and Valberg, 1986).

The committee hoped that because of the striking and unusual colour relationships displayed in its various colours scales, "artists and designers [would] find it useful in devising new and beautiful arrangements of colors" (MacAdam, 1974). In fact, that has happened. Artist Karl Gerstner, for example, has made use of this system in his art and states

> Whether the *uniform color space* is the final model of colorimetrics, sought by so many generations of scientists, can be determined only by future research. As far as I am concerned, one thing is certain: it imparts a surprisingly novel experience of color and could not be more stimulating to the artist (Gerstner, 1986).

2.4 Artistic Colour Use

To create a window system that assists with aesthetic colour selection, it is useful to consider the techniques used by artists to select harmonious colour. Many artists have developed theories of colour, including (Albers, 1971), Gerritsen (see Section 2.2.6), (Itten, 1961), (Munsell, 1947) and many others. For a good introduction to such theories, see (Hope and Walch, 1990) and (Norman, 1990).

Recently, artist Stephan Quiller has taken the standard colour circle, familiar to all artists, and developed a detailed colour model for use by artists (Quiller, 1989). It is based on the subtractive colour circle that all

art students learn; the primaries are red, yellow and blue, the secondaries are green, violet and orange. Quiller explains how to use the model effectively, especially by describing techniques for creating harmonious colour palettes using a few simple colour schemes.

While much of Quiller's discussion applies only to painting, the concepts of colour harmony he presents are generally applicable. He describes five major colour schemes that can be created in a structured manner using the colour wheel and discusses how to create harmonious colour schemes with them.

The Monochromatic Colour Scheme The simplest of the colour schemes uses a single hue, which can be varied in both lightness and saturation. When using a monochromatic colour scheme, the key issue is how lightness and saturation are used. For example, since the eye is drawn to areas of strong contrast they should be used sparingly and purposefully to attract the eye; otherwise, "the eye would be disturbed and would have a hard time looking at the painting."(Quiller, 1989, p.31). Areas away from the center of attention should use colours which are similar in lightness and saturation.

By their very nature, the colours of a monochromatic colour scheme are harmonious. While it is hard to create visually stunning effects with this scheme, the results are usually pleasant, if somewhat bland.

The Complementary Colour Scheme Complementary colours are located opposite each other on the colour wheel (or hue circle.) When true complements are mixed together, they produce "beautiful neutral colours" (Quiller, 1989, p.38). When mixed in correct proportions, they produce a neutral grey. The complementary colour scheme, therefore, consists of a colour, its complement and all the semi-neutral colours created with them.

The complementary scheme works especially well because of the visual phenomenon of simultaneous contrast (see Section 2.5.) However, it is precisely this effect which makes this scheme hard to use. If both the highly saturated complementary colours are used equally throughout a composition then they compete. One way of using two complementary colours effectively is to pick one as the dominant colour and the other as subordinate. The dominant colour is used in large amounts with varying value and intensity. The subordinate colour is used in small amounts to bring the dominant colour to life. Quiller does not give a satisfactory explanation of how to use dominant and subordinant colours, instead relying on examples and exercises to help the artist develop a feeling for the correct use of this scheme.

The Analogous Colour Scheme The analogous colour scheme uses three closely related colours. By considering the twelve primary, secondary and tertiary colours, any adjoining three of these (plus the light and dark values and semi-neutrals of those colours) make up an analogous colour scheme.

Like the monochromatic colour scheme, the analogous colour scheme is harmonious by its very nature. By selecting three adjoining colours, each colour is close enough to the others to be visually related. Analogous colour schemes are slightly harder to use than monochromatic ones, but more visually interesting.

The Split-Complementary Colour Scheme The split complementary colour scheme is a union of the complementary and analogous schemes. First, select the analogous colour scheme that is to dominate the composition, then add the complement of the middle analogous colour to the scheme. This scheme is a natural extension of the analogous scheme, with richness added by the complementary colour.

While giving a richer result than the analogous scheme, this scheme shares the problem of the complementary scheme; it is difficult to achieve balance between the complementary colours, the choice of saturation and value being a very subjective decision.

The Triadic Colour Scheme The final colour scheme presented by Quiller, the triadic colour scheme, is created by selecting any three equally spaced colours around the circle for the primary colours. This scheme creates very colourful and potentially striking palettes, but is much more difficult to use than the above mentioned schemes. Satisfactory use of it seems to depend on sophisticated artistic intuition.

What makes these colour schemes important is their applicability to be algorithmic application. The monochromatic and analogous colour schemes are especially useful because of their simplicity and inherently harmonious nature. The complementary, split-complementary and triadic schemes require significantly more artistic intuition to be successfully applied, and thus do not lend themselves as well to automatic application.

2.5 Colour Perception

The psychophysical and physiological basis of human colour vision, the process by which coloured light is converted into an image inside the human brain, is far from being completely understood. For an in-depth look at colour vision, see (Boynton, 1979).

Some things are known about colour vision: for example, it is believed that the colour information in the visual system is processed in parallel and independently of other aspects of vision (Cowan and Ware, 1985). Also, the sensation of colour hue is very complicated, being far more than just a perception of the dominant wavelength of the light entering the eye. Josef Albers (1971) shows how simple it is to create visual illusions such as making two colours appear as one. Other visual illusions, such as creating impressions of transparency and depth with colour, are discussed by Albers and in (Hope and Walch, 1990).

Many of the effects discussed by Albers are the result of simultaneous contrast, an effect where colour differences are increased at edges. Other aspects of colour perception, such as successive contrast, colour blindness and chromatic aberration, also affect the appearance of colours. What is important for the reader to realize is that colours interact with other colours in proximity to them, altering the viewers perception of their colour value. A few perceptual effects are especially relevant to this thesis, and are discussed below.

Colour Constancy An extremely important perceptual phenomenon is *colour constancy*. This is the effect whereby different objects maintain their colour appearance under a wide variety lighting conditions. The easiest way to see the effect of this is to put on a pair of coloured sunglasses on a sunny day. Immediately, all the colours in the visual field will change due to the loss of light and the colour of the glass. After a short time, the objects being viewed will seem to change back to their original colours, even if the sunglasses are tinted a distinctive colour such as red or orange. Similarly, human vision is capable of accommodating huge ranges of brightness, from bright daylight to moonlight. With very dim light, however, the sensation of colour ceases, with red being visible the longest as light is decreased.

Veiling *Veiling* is a situation whereby the lighting conditions in the viewing area alter colour appearance. With respect to self luminous objects, such as CRTs, the spectral power distribution V_λ of a veiling light is added to the object's spectral power distribution E_λ, changing the colour of the object. Consider the colour matching experiment described in Section 2.2.2. To change the appearance of one coloured light, another coloured light is added to it. Similarly, if the amount of light in the viewing area of a coloured light source is changed, the colour of that light will change. This effect is seen when the lights in the area of a computer screen are turned on, or the sun shines on a computer screen. In either of these cases, the addition of large amounts of white light causes the colours on the screen to appear lighter and desaturated.

Colour Memory Human *colour memory*, or the ability to remember a specific colour and be able to identify it later, is not very good. The concept of colour memory, as used here, is best illustrated with an example. Consider the colour red used on stop signs. This is a colour that most people have seen many times. Yet a typical observer, given a group of similar reds, one of which is the red used on stop signs, would most likely not be able to identify it.

Helson et al (Helson, Judd and Warren, 1952) studied object colour changes under different lighting conditions. In the course of their study, they trained subjects to identify colours in the Munsell colour space. Even after training, their subjects were not able to exactly identify the Munsell name of a sample colour. Their results were informally analyzed and show that the subjects were only capable of identifying approximately 1500 distinct colours. If this is all that can be expected when subjects are actively trained to identify colours, much less can be expected of the average untrained person, perhaps as few as 100-150 distinct colours.

2.6 Basic Colour Terms and Colour Discrimination

The linguistic concept of *basic colour terms* were introduced by anthropologists Berlin and Kay (Berlin and Kay, 1969). It has been suggested, based on their work, that there may be only 11 categories of basic colour sensations, each associated with a well-learned name and possibly a unique physiological substrate. Basic colour terms provide faster and more reliable colour naming than other terms, with greater agreement amount viewers. (Smallman and Boynton, 1990). Colour naming is important because colours on a display will often need to be cross referenced, as in "the red window" or the "green window with blue text". Furthermore, if the suggestion that each of these basic colour names represents a basic physiological colour sensation is correct, these colours should be optimal for colour categorization.

There is no simple definition of what constitutes a *basic colour term.* Intuitively, they are words such as *black, white,* and *green* but not expressions such as *crimson, scarlet, blond, blue-green, bluish, lemon-coloured, salmon-coloured,* and *the colour of the air pipes in the Davis Centre.* Berlin and Kay (Berlin and Kay, 1969) proposed that the basic colour terms are similar across all languages. This proposal contradicted the prevailing doctrine of American linguists and anthropologists, which held that each language performs the linguistic coding of life experiences in a unique manner. They found that, although different languages encode in their vocabularies different numbers of basic colour categories, a universal list of eleven categories exists from which the basic terms in all languages are drawn. These terms are *black, white, red, green, yellow, blue, brown, purple, pink, orange,* and *grey*. Their findings have been experimentally confirmed by Boynton and his students (Boynton and Olson, 1990; Uchikawa and Boynton, 1987; Uchikawa, Uchikawa and Boynton, 1989; Smallman and Boynton, 1990).

The last of these studies is of particular interest. Prior to this paper, research seemed to suggest that there is an upper limit on the number of colours that can segregate successfully, perhaps as low as six. In the context of a computer display with coloured windows, this implies that if more than six different colours are used for the windows, the ability of the user to organize these windows and quickly associate a certain colour with a certain window context is significantly diminished. However, Smallman and Boynton showed that by drawing colours from the eleven basic groups, this limit can be almost doubled. Furthermore, the use of basic colours yields equivalent search performance for two types of cues—examples and names. This is an important point, because coloured objects are often referred to by colour names. They also noted that the good separation of the basic colours[5] counts more for useful segregation than the fact that they are basic colours. The mean interpoint distance between each of the optimal basic colours and all others is shown in Table 2.3. By using a set of non-basic colours with only half the interpoint OSA distance of the basic colours, Smallman and Boynton showed that in

[5]The separation of two colours is defined here as the Euclidean distance between them in the Optical Society of America (OSA) uniform colour space.

Basic Colour	Mean Interpoint OSA Distance
Yellow	13.84
White	13.01
Orange	12.73
Blue	12.14
Green	11.59
Pink	11.37
Purple	11.34
Black	11.30
Red	11.04
Brown	9.47

Table 2.3: Mean Interpoint Distance in the OSA Uniform Colour Space is OSA Units between Each Optimal Basic Colour and All Others (Smallman and Boynton, 1990)

one extreme case, efficient colour coding can take place with 14 colours as long as they are well separated in the OSA space. However, the use of basic colours is still advantageous because the ease of attaching universal names to the colours for cross referencing.

The suggestion that colour distance accounts for the segregation of two colours raises an interesting question. It is well known that adaptation in the human visual system causes sensitivity to luminance contrast to scale relative to the range of the luminances of all objects in the visual field, dulling sensitivity to small contrasts when large contrasts are present (Cowan and Ware, 1985). Does a similar effect hold for chromatic contrast? If so, similar colours can be easily differentiated as long as there are no radically different colours present in the visual field. However, using closely spaced colours is often a bad idea because of the poor quality of colour memory. For example, consider a display that uses three shades of blue. While all three shades are visible, it may be quite easy to discriminate between them. However, if the darkest shade is removed, it is hard for a user to decide if the darker of the two remaining colours is the “darkest” of the three, or the “middle” of the three. This is because relative judgements are precise, so the user will create associations with the “dark blue”, the “light blue” and the “middle blue”. When one is removed, the relationship is destroyed. When the display consists of three basic colours such as red, blue and green, removing one colour does not pose the same problems.

In addition, the effects of variable illumination and the large variance in colour response of different monitors means that colours which were designed to be close to one another, such as similar shades of some colour, may sometimes appear the same.

Grouping of colours using the basic colour terms satisfies a key requirement for organizational colour use: they are easily distinguishable from each other by the vast majority of readers and they have unique names.

2.7 Contrast and Reading

When using colour for text, the most important consideration is readability. It is irrelevant how aesthetically pleasing the colours are if they prevent the text from being read! Legibility has been addressed by numerous studies with quite similar results (Legge, Rubin and Luebker, 1987; Legge, 1989; Legge et al., 1990; Rubin and Legge, 1989; Tinker, 1963; Knoblauch and Arditi, 1989; Knoblauch, Arditi and Szlyk, 1990; Gould et al., 1987).

Many factors influence the legibility of text displayed on a computer screen: the typeface of the text, the resolution of the computer display, the contrast between the text and the background, and so on. Contrast is particularly interesting. Most studies examine *luminance contrast* (the luminance difference between the characters and the background) without considering *colour contrast* (the chromaticity difference). Normally, the clarity of text is a function of the luminance contrast between the text and the background. However, luminance contrast is not required for text to be readable. Colours with the same luminance (referred to as equiluminant colours) but with high chromatic contrast can create legible text (Knoblauch, Arditi and Szlyk, 1990; Legge et al., 1990).

Luminance contrast is usually measured by comparing the luminance of the foreground and background colours[6] using a standard contrast measure, such as Michelson contrast.

$$C = \frac{L_{max} - L_{min}}{L_{max} + L_{min}} \tag{2.15}$$

C ranges from 0 to 1.0. Unfortunately, a reasonable measure for equivalent chromatic contrast is not quite as straightforward. While measuring the distance between two colours is possible in a uniform colour space, it is not obvious how to measure contrast using these distances. Legge (Legge et al., 1990), for example, measured chromatic contrast by looking at the wavelengths of light seen by the long- and medium-wavelength-sensitive cones in the eye.

The major results of the aforementioned studies are as follows:

- For people with normal vision, reading speed deteriorates by about half as luminance contrast falls from 100% to 12%. Below 12% contrast, reading rate drops very rapidly. It appears that for contrasts greater than approximately 12%, reading rate is not significantly affected.
- Chromatic contrast is similar to luminance contrast, *once the data is normalized to the respective threshold*[7] *levels*. To further complicate matters, the reading threshold changes based on the character size. At normal to large text sizes, consisting of characters with a visual angle[8] of less than 1°, the luminance threshold is lower than the colour threshold. However, when character size increases to very large sizes of around 6°, the colour threshold lowers to levels approximately equal to the luminance threshold. It is possible that chromatic contrast may outperform luminance contrast for character sizes larger than 6°.
- The two forms of contrast are processed independently within the visual system. It appears that readers rely on whichever conveys the most information. Readability is equivalent to the maximum of luminance and chromatic contrast.

With the exception of Gould et al. (1987), these studies look at reading speed for relatively small amounts of text, with no consideration given to reading comfort or effort required. Clearly, reading at maximum speed should require little effort, or extended reading will be fatiguing and quite unpleasant. Intuitively, one would expect a range of contrasts where reading at maximum speed is possible, but where the reader has to strain somewhat to achieve this speed. This situation is not desirable from the point of view of the reader, even if they are not consciously aware it is occurring. The issue of reading comfort has been addressed by (Gould et al., 1987). His interest is in why reading from a CRT is slower than reading from paper. Unfortunately, his suggestions for improving reading speed on a CRT do not consider colour or contrast, rather concentrating on issues such as character size while using only black on white or white on black text.

[6] The luminance of a colour can be measured using a photometer, for example.
[7] The reading threshold refers to the minimum level at which reading is possible.
[8] visual angle is "the angle formed at the eye by rays from the extremities of an object viewed"(Allen, 1990)

In section 2.6, several issues are discussed which relate to the way colours are perceived. Many things that can change the appearance of a colour, such as lighting and vision problems, are considered. Just as these factors need to be taken into account when quantifying colour discriminability, they should be considered when discussing levels of contrast. The luminance of a colour, and thus its contrast with any other colour, is directly affected by such things as lighting conditions. By increasing the background light in the vicinity of a CRT, or more drastically by shining light directly on the CRT, luminance contrast is diminished. Similarly, the perceived saturation of the colours will be decreased by strong veiling lights, reducing chromatic contrast.

Adaptation presents another interesting problem because the threshold level for contrast is dependent on the largest contrast presented to the reader.

2.8 Colour Harmony

According to the Concise Oxford Dictionary (Allen, 1990), harmony is defined as "an apt or aesthetic arrangement of parts ... the pleasing effect of this." Unfortunately, defining harmonies, particularly colour harmonies, is not simple. Meier (1988) looked at the problem of colour harmony and reported

> We tried to discover a general relation between any two colors in a three-dimensional color space that would show whether the two colors harmonized or contrasted and how attractive they appeared together ... we were unable to find any general relations.

Similarly, Hope and Walsh (1990) comment

> Harmonies are subjective; those that appeal to some people, repel others. Although the human eye and mind are sensitive and efficient in sorting out, responding to, and creating harmonies of color, it has proven impossible to formulate and establish absolute rules for harmony.

There are many theories of colour harmony, but most of them share a few common problems. First, they are usually created for a specific audience. For example, Quiller's (1989) theories (Section 2.4) are directed at artists and Albers (1971) theories (Section 2.5) are directed at the graphic designer. Second, they tend to be medium specific. Quiller discusses the number of pigments are use, selecting colours for use as translucent washes, the effects of mixing pure pigments with black and white, etc. Albers, on the other hand, discusses the relative sizes of pieces of coloured paper, visual illusions such as simultaneous contrast and how to avoid them, etc.

Finally, and most importantly, none of these theories are sufficiently prescriptive or rigorous. They all rely on the colour sense of the user to such a degree that they are not easily computerized; artists and designers cannot provide algorithms for harmonious colour selection. This inability occurs because many guidelines are expressed in terms of such concepts as "visual balance," which are possible to explain through examples but not with algorithms. These theories are necessarily underdetermined, and must be to allow for artistic expression. Consider the following:

- Given the same theory of colour harmony, two designers would most likely create two completely different colour palettes, each of which fits the constraints of the given theory.
- It is possible to choose palettes that satisfy all the guidelines of a theory, yet are not aesthetically pleasing.

This is not to say that the quest for colour harmony is hopeless. The ultimate model of colour harmony is not in sight, but some modest results are possible and, for our requirements, preferable. For example, two

of Quiller's colour schemes, the monochromatic and analogous schemes, seem convertible into algorithms that generate harmonious colour palettes. While uninteresting from an artistic viewpoint, these palettes will provide some minimum level of harmony that will be inoffensive to many people. While the "best" harmonies are novel or unexpected, they are the least easy to find by algorithmic approaches and the most likely to arouse disagreement. Algorithms for generating harmonious colours are a good subject for a thesis, but are not the main subject of this thesis.

Many colour theories aim to develop the student's sense of harmony, as opposed to dictating rules of colour use. By teaching and suggesting instead of dictating, more of the historical colour theories can be used. Thus, while many of the theories cannot be turned into algorithms that always generate harmonious colours, they may be turned into algorithms that *sometimes* generate harmonious colours. By incorporating human criticism with computer generation, these algorithms are useful.

Chapter 3

Colour Usage

This chapter explores how colour is used, both in a general way and in computer window systems specifically. Unfortunately, use of colour is not a well understood topic. Despite much research, results in the area amount to little more than conjecture, rules of thumb and subjective guidelines. The aim of this chapter is not to create a comprehensive model of colour usage, but to extract a set of theories that can provide a realistic basis for a set of tools to assist in the task of colour selection for windowing systems.

First, a useful categorization of colour usage is explained. Following this, important uses of colour in window systems are described in the context of this categorization, both to show its utility and to highlight the problems this thesis will attempt to solve.

In this chapter, a "colour" refers to colour as the property of the appearance an object, such as the background colour of a window or the colour of an apple. "Colour value" refers to the location of a colour in one of the many colours models described in Section 2.2. Thus, text in a window has a colour which in turn has a colour value with respect to one of the colour models.

3.1 Categorization of Colour Usage

Colour is one of the most poorly understood aspects of computer window systems. It is rare to find two users who describe their colour use the same way. For example, some users describe using colour for purely aesthetic reasons, while others mention using colour to distinguish different windows. Taking these descriptions as a whole, however, it is possible to distinguish several common themes.

Colour usage can be divided into four categories by distinguishing between two important aspects of colour usage, colour relationships and colour intent, as shown in Figure 3.1. The first, colour relationships, divides colour usage between *absolute* and *relative*. The distinction is between colours whose values are determined from an absolute specification and colours whose values are determined relative to the values of other colours. The second, colour intent, divides colour usage between *functional* and *aesthetic*. This distinction is between colours whose values are intended to subserve a utilitarian function and those whose function is aesthetic.

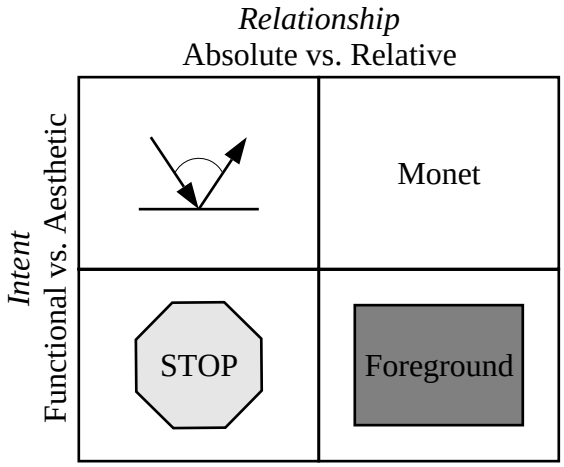


Figure 3.1: Colour usage can be divided into four categories by distinguishing between colour relationships and colour intent. Typical examples are raytracing, the Impressionist paintings of Claude Monet, stop-sign red and adequate foreground/background contrast.

3.1.1 Absolute versus Relative Colour

3.1.1.1 Absolute Colour

Absolute colour refers to colour values that are meaningful because of their absolute position in the colour space. For example, red is often used to signify danger or warning, as when used on a stop sign. The meaning associated with red is created by society and is linked to the absolute position of the colour value. Another use of absolute colour occurs in the use of trademarked colours. The blue used by Xerox is different from the blue used by IBM, which is in turn different from the blue used by Sun. The exact blue colour in each case is trademarked by its absolute colour value.

When realistic images are created using techniques such as ray tracing or radiosity, colour is used in an absolute way. The exact colour value for each point in an image is computed by simulating the interaction of light and matter. Thus, the meaning of the colour values are related to their absolute position in colour space. This is an example of the absolute use of colour.

3.1.1.2 Relative Colour

Relative colour, on the other hand, refers to colour values which are meaningful through their relationships with other colour values. Consider a user of a computer window system attempting to select colours for many windows. Aesthetic considerations notwithstanding, the user will consider certain criteria when selecting values for these colours, some of which represent functional colour usage. For example, there will need to be enough contrast between the foreground and background colours of any particular window so that the window contents to be legible. More interestingly, the values of colours in unrelated windows will need to be distinct enough that the windows will appear to be visually unrelated. In both of these cases, the absolute values of the colours are not important. Rather, their values relative to other colours are at issue.

Consider the way in which certain impressionist and postimpressionist painters, such as Van Gogh and Monet, used colour. Instead of using purely realistic colour values, they used relative colour techniques to create more vibrant and intense colour impressions. By exaggerating the colour values in the shadows of a painting, for example, a viewer's perception of the colours outside of the shadows can be enhanced, possibly beyond the limits of the paint gamut. Another technique used by some impressionists was to mix dabs of bright colour directly on their canvases and allow the viewer's eye to mix these colours, creating additive colour sensations in a subtractive medium. In both of these examples, the actual colour values being used are important only in their relation to the other colours in the paintings; as such, they represent relative colour use (Hope and Walch, 1990).

Obviously, there are cases when the dividing line between absolute and relative colour is not clear. A particular shade of red may be chosen both to represent danger and to provide enough contrast with another colour to allow text to be read. In this situation, the colour has both absolute and relative properties. Which is most important varies from one situation to another. However, both properties exist and both are used to determine the value of the colour, regardless of which is more important in the mind of the person selecting the colour.

3.1.2 Functional versus Aesthetic Colour

3.1.2.1 Functional Colour

Functional colour refers to colours whose values are chosen to provide some sort of functional benefit to the user. This includes, for example, colour used to imply *professional association* through the use of trademarked colours, to convey spatial, temporal and structural *organization*, to aid in *comprehension* or to *emphasize* important information.

Consider the examples of relative colour described in the preceding section. The problem of choosing visually unrelated colour values for logically unrelated windows represents a functional use of colour because the logical organization of the windows is conveyed to the user by colour. A more obvious use of functional colour is the selection of colour values for the foreground and/or background of a window. In this case, the colour values must contrast enough that the contents of the window are legible. Again, the values of the particular colours are chosen to provide some functional benefit to the user.

This is not to imply that functional and relative colour are equivalent. For example, the absolute use of the colour value red to signify danger is also an example of functional colour usage.

3.1.2.2 Aesthetic Colour

Aesthetic colour refers to colours whose values are chosen for their aesthetic properties. Very few colour choices are made without considering aesthetics to some degree.

Users often believe that colourful interfaces improve their productivity. They feel that colour is less monotonous and causes less fatigue and eyestrain (Meier, 1988). Colour workstations are often seen as a status symbol or an indication that their employer is concerned about them. In addition, colour allows users to personalize their environment by expressing their colour preferences, which may make the environment more pleasant to use. If the environment is more enjoyable to use, people may work longer and take fewer breaks. Conversely, if the colours used by the system are arbitrarily restricted and the user finds them unpleasant, they will not enjoy using the environment which may have detrimental effects on their productivity. There are many sets of colours that satisfy the functional requirements of the window system. The user should be allowed to select a set that they prefer.

As with the boundary between absolute and relative colour, the distinction between functional and aesthetic colour is not always clear. As illustrated above, when a colour is selected for use as a background colour in a window, there are both aesthetic and functional aspects to be considered.

3.1.2.3 The Fallacy of Functional and Aesthetic Incompatibility

There is a belief among many people that aesthetic and functional considerations are at odds with one another. Statements such as “Any decorative use of colour should be subservient to the functional use” and “In general, it is good to minimize the number of different colours being used” are common in guidelines for user-interface design (Foley et al., 1990). The purpose of such guidelines is to avoid garish displays and unintentional user associations.

Consider the implications of allowing users to choose the foreground and background colours for their windows. In the previous section it was suggested that there are many sets of colours that satisfy the functional constraints of a window system, so the user should be allowed to select the set that they prefer. However, there is no guarantee that users will select colours that satisfy any of these functional constraints, such as provide enough contrast to enable them to read quickly and with a minimum of errors. This is obviously of concern to employers, since productivity can be adversely affected by such problems.

However, the problem is not that functional and aesthetic constraints are incompatible, but rather that there are no tools available to help the user select colours in an intelligent manner. When users make colour choices, they do not purposefully select colours which violate the functional constraints. Instead, they are either unaware of all the functional constraints that should be considered or incapable of creating a set of colours that satisfies these constraints. Even if the choices are being made for purely functional reasons, given a reasonably large number of windows, it is quite difficult for most people to satisfy all the constraints without help.

3.1.3 Multiple Categorization

Most colours on a display can be described by more than one of the four categories discussed above. The categories describe the uses of colour, and most colours are determined by the interaction of more than one functional or aesthetic requirement, either absolute or relative. For example, the choice of the colour used for the title on the cover of a book may be determined by any of the following:

- the relative contrast between the background and foreground colours.
- the absolute colour associated with the publisher, book series, author, institution or company.
- the aesthetic interaction of the title with pictures on the cover.
- the aesthetic judgments of the publisher, author or cover designer.
- contrast with other books on the same topic.

Therefore, when considering colour usage in window systems it must be remembered that most colours are simultaneously subjected to constraints from more than one of the four categories. As as result, it is not always possible to identify a primary use for any colour, nor is it necessary to do so.

3.2 Colour Usage in Window Systems

The next step in developing a set of tools to assist with colour selection is to examine how colour is used in window systems and decide how the window system can help the user. Each of the four categories introduced above are discussed in turn, highlighting the common ways colour is used in each of them.

3.2.1 Absolute Functional Colour Use

Absolute functional colour is commonly used in window systems to create an *association* between the colour and something. Consider the following:

- Colour denotes a physical property of an objects, such as ripeness.
- Colour denotes a conceptual property of an object, such as a blue being associated with IBM or red being associated with danger.

There are two important things to note about absolute functional colour use. First, there are obviously times when functional requirements of colour demand that the colour value displayed be exact, such as is the case when a trademarked colour is used. Conversely, there are times when a request for an absolute colour does not require an exact colour value. Consider using red to associate something with danger. There are many shades of red and many of them serve the purpose equally well. Both of these cases of functional colour must be supported by any usable system.

3.2.2 Relative Functional Colour Use

Relative functional colour can be used in many different ways in computer window systems. Two of them, ensuring *legibility* and showing the logical or structural *organization* of the windows, are particularly important.

Ensuring legibility is the most basic functional requirement. In studies conducted by Legge and others (discussed in Section 2.7), reading speed dropped when contrast was inadequate, and reading errors also increased. As conditions deteriorate in these studies, so does reading performance. However, this could be due to speed, errors or subject fatigue. Gould et al (1987) showed that even with the maximum possible contrast, reading on a CRT is usually slower than reading from paper.

Structural organization can be effectively conveyed through colour. As a simple example, similar colours can be used to display functionally related windows and distinct colours to display unrelated windows. A major concern in selecting colours in this manner is to avoid false associations. This problem, unfortunately, is not easily solved. Colour difference is relatively easy to measure (see Sections 2.2.2.2, 2.3 and 2.6) but there are additional factors affecting colour similarity which are not understood and make it more than just the inverse of colour difference. For example, the distance between a particular value of red and and some shade of pink may be exactly the same as the distance from the red to some shade of yellow, yet there is something about the red and pink that makes them more strongly associated. In this case, the similarity of their hues probably accounts for the association between them. However, consider a red and a pink that have the same difference measure as a yellow and a brown. In this case, the yellow and brown may also have the same hue, but the association between them is much weaker. Alternatively, consider the situation in which the majority of the colours in the users visual field are very desaturated. Those few colours that are saturated may be associated by the user. The key phrase here is *may be associated*, as people create associations differently. Associations between colours are also created for many other reasons, such as the nationality and age of the user, that vary from person to person and cannot possibly be accounted for. As a result, it may be impossible to guarantee that false associations are never created on an extremely colourful display.

Other uses of relative colour will not be discussed in depth. For example, colour is often used to aid in *comprehension* and to *emphasize* important information, both of which represent relative functional colour use. In both cases, the problem can be thought of as showing the logical or structural relationships of objects *within* a single window or application. The problems are analogous to the problem of window organization, and can be handled similarly.

3.2.3 Absolute Aesthetic Colour Use

Absolute aesthetic colour, as can be inferred from the above, refers to any colour whose aesthetic properties depend on its actual colour value. Examples of absolute aesthetic colour usage are the following:

- Colours that are required to be an exact value for some artistic or subjective reason. This is the typical interpretation of absolute aesthetic colour usage.
- Colours whose values are precisely computed as part of some image generation technique, such as ray tracing, and whose values must be displayed exactly as specified. It should be noted here that while the term aesthetic is usually associated with artistic colour usage or personal colour preferences, the colours used in pictures generated by realistic rendering techniques are also considered aesthetic under this categorization. The individual colour values serve no function aside from creating a realistic image. The success of the image is determined by the realism of its appearance, which is largely an aesthetic issue.
- Colours that can have one of many values within a certain absolute range. Consider, for example, when a user decides that a certain window should have a green background. In some cases, an exact value of green is required, but more often the desire of the user is to have a colour "something like a certain green". The exact value is not important. Often the specific value of the colour is determined by the user in an attempt to solve some other functional or aesthetic issue.

Just as with absolute functional colour, different degrees of *absolute* are required. Colour will need to be specified in any level from "exactly this value" to "something close to this value."

3.2.4 Relative Aesthetic Colour Use

Most of the colour choices that a user makes are determined to some degree by aesthetics. As a result, relative aesthetic is the category of colour use employed most often by users of window systems. While some aesthetic colours choices are absolute, many more are made in relation to existing colours, since the easiest way of achieving harmonious colour is to select colours that have a close relationship to each other. Consider the colour schemes suggested by Quiller in Section 2.4. All of these schemes are defined by one, two or three absolute colours. All other colours in a composition are chosen relative to these.

Chapter 4

Colour Contrast

The major obstacle when attempting to select colours is the large number of constraints that simultaneously affect any single colour choice. There are two ways a system can assist users in making effective and harmonious colour choices:

- the system reduces the number of constraints that must be explicitly handled by the user. In particular, to enable users to concentrate on aesthetic issues, the system should reduce the number of functional constraints that the user must consider.
- the system reduces the number of choices the user must make. In particular, the user is allowed to make a few aesthetic colour choices and the system selects the remaining colours to satisfy functional and aesthetic constraints.

In both cases the system must make colour choices based on functional colour constraints. Maintaining sufficient contrast between background and foreground colours to provide legible and readable window contents is the most important functional colour constraint. In this chapter a metric for measuring the contrast of a pair of colours is presented.

There has been much work done relating to contrast and reading, as discussed in Section 2.7. When considering the effects of contrast on colour displays, two aspects of contrast must be considered. *Luminance* contrast is the relative difference between the luminance of the foreground and background colours. *Chromatic* contrast is the relative difference between the chromaticity of the foreground and background colours. Research has shown that either luminance contrast or chromatic contrast is sufficient for reading and legibility, and that the visual system will use whichever one provides the greatest legibility. However, there is no generally accepted metric for measuring chromatic contrast. For this reason, only luminance contrast is considered here.

4.1 Luminance Contrast

Luminance contrast measures the difference in lightness of the foreground and background colours, usually as a ratio. Different studies of legibility and contrast find a wide range of values for the minimum contrast consistent with satisfactory reading, probably because of variations in the meaning of “satisfactory”. For example, both Legge (1987) and Knoblauch (1990) find deterioration in reading with contrast to be statistically significant only when the contrast drops as low as 12%. By this point, however, reading speed is down to 50% of the high

contrast value and reading comfort probably much lower. Gould et al (1987), on the other hand, found reading speed on CRTs to be lower than paper reading speed at all CRT contrast levels. An informal study conducted by the author found luminance contrasts between 80% and 95% required for comfortable reading by some users[1]. As a result, the minimum contrast value should be user adjustable, with a value around 12% being the absolute lowest reasonable value and something much higher, such as 80% or more, used as a reasonable default. Because luminance contrast between colours with similar chromaticities may need to be very high for comfortable reading, chromatic contrast should be studied further. Otherwise, many pairs of colours with low luminance contrast but high chromatic contrast will be disallowed even though they may be perfectly acceptable.

4.2 Calculating Luminance

Before Equation 2.15 can be used used to calculate the contrast between the background and foreground colours, the luminance of these two colours must first be determined. Section 2.2 shows that the luminance of any colour on a display is the Y coordinate when the colour's value is expressed in the XYZ colour space. To convert from one of the standard colour models used in computer graphics to XYZ, the chromaticity coordinates of the display must be known. They are often available from the monitor manufacturer, or can be measured with a colorimeter as described in Section 2.2.2.2. If it is not possible to determine the chromaticity coordinates of the display, the coordinates of a similar monitor or the coordinates of a standard monitor can be used. However, while many monitors have similar coordinates, they do vary, even between similar models. Large variations are probably due to some fundamental change in the internals of the monitor. Such changes occur even between monitors of the same make and model. Therefore, if the chromaticity coordinates of the monitor are not available for the conversion to XYZ, the contrast threshold should be raised to provide an engineering margin of error. Some colour combinations are then eliminated but not enough to make it worth running the risk of producing illegible text. Alternatively, a system could be developed for interactively adjusting the default chromaticity coordinates to more closely correspond to the monitor. However, there are far more serious issues that affect the effective luminance of the colours.

The Y values of the foreground and background colours are proportional to the intensity of light being *generated* by a given pixel. This is not sufficient to calculate contrast because the *measured* luminance of any pixel on the display surface is affected by other factors (Cowan, 1989; Klassen, 1989). First, the ambient lighting in the room can affect the measured luminance, especially for dark colours. Second, a monitor's *black level*, or the luminance of black relative to the luminance of white, is not zero but some small amount greater than zero. Most monitors are equipped with two controls, a *brightness* control and a *contrast* control which greatly affect the black level of the monitor (see the discussion of gamma correction in Section 2.1.1). Third, light emitted by a pixel affects the neighbouring pixels, a effect called *pixel bleeding*. The shape of a single pixel is generally taken to be Gaussian. Therefore, the light contributed to a neighbouring pixel is proportional to the exponentiated inverse of the square of the distance between the pixels, as shown in Equation 4.3. Since contrast is a measure of the difference in intensity between pixels along edges of luminance or colour difference, the fact that the intensity of a pixel is affected by the intensity of those around it must be considered.

4.2.1 Ambient Lighting and the Brightness Control

Without knowing both the lighting conditions in the viewing area and the setting of the brightness and contrast controls, it is impossible to determine the exact brightness of any pixel. As discussed in Section 2.1.1 the purpose

[1] These number represent a percentage of the maximum contrast available, using approximations developed in this chapter. For example, if the maximum possible contrast is 50%, users required values between 40% and 48%.

	Monitor A		Monitor B	
	Typical Light	Bright Light	Typical Light	Bright Light
Luminance of White	14.01	15.50	6.45	8.51
Luminance of Black	0.17	2.11	0.05	1.75
Black Level	0.01	0.14	0.01	0.21
Contrast	0.98	0.76	0.98	0.66

Table 4.1: The luminance of large areas of black and white were measured on two typical monitors without adjusting the brightness or contrast controls. Monitor A is a Sun colour monitor Model GDM-1662B, manufactured by Sony. Monitor B is DEC monochrome monitor Model VR 260-BB. The viewing area is a computer lab equipped with both dim incandescent directional track lights and overhead fluorescent lights. The typical lighting condition occurs with only the dim incandescent lights turned on. The bright lighting conditions occur when the fluorescent lights are turned on as well. The typical low lighting conditions are representative of many computer labs. The contrast and brightness of the monitor were not adjusted to account for the brighter light. The contrast was calculated using Equation 2.15.

of having brightness and contrast controls on a monitor is to allow the user to respond to changes in ambient illumination. The black level of a correctly set-up monitor is black, allowing us to ignore the effects of ambient lighting on the black level.

If the phosphor chromaticities, black and white levels of the monitor are measured, they should be measured in typical viewing conditions so the typical ambient light and monitor adjustments are taken into account. This is important, especially for the black and white levels. Table 4.1 shows the effect of different lighting conditions on the luminance of large areas of black and white on two typical monitors. The monitor was not re-adjusted to account for the bright light. As can be seen, ambient light drastically increases the black level of the monitor if the brightness and contrast controls are not adjusted. The result is a dramatic decrease in the contrast at a black and white edge. If the black level of the monitor cannot be measured at typical light levels, a reasonable default is to use 1% of the maximum white level, as is the case with both of the monitors in Table 4.1 when the controls are adjusted for the typical lighting conditions (Cowan, 1989).

The black level of the monitor represents the minimum luminance that can be produced on the monitor. To account for this, the black level should be added to the calculated luminances of the colours before the contrast is calculated. If Y_{fg} and Y_{bg} are the Y coordinates of the foreground and background colours, the luminances are

$$L_{bg} = Y_{bg} + L_B \tag{4.1}$$
$$L_{fg} = Y_{fg} + L_B \tag{4.2}$$

where L_B is the black level of the monitor.

4.3 Pixel Bleed

The problem of pixel bleed is more difficult because it cannot be ignored and is very difficult to measure. The contribution of any pixel to its neighbours varies from monitor to monitor, but is significant in all cases. For a field of full intensity pixels, almost 40% of the brightness of an individual pixel can be contributed by its neighbours.

4	4	3	4	4
4	2	1	2	4
3	1	**P**	1	3
4	2	1	2	4
4	4	3	4	4

Figure 4.1: The Neighbours Of Pixel **P**.

The spatial profile of the light emitted by a given pixel can be expressed as a Gaussian function centered at the pixel location (x_0, y_0) by[2]

$$\Phi(x, y) \propto \exp\left(\frac{-((x - x_0)^2 + (y - y_0)^2)}{\sigma^2}\right) \tag{4.3}$$

The amount a pixel influences others around it depends on the value of the constant σ, which represents the amount of pixel bleed[3]. As was mentioned above, this value varies from monitor to monitor. Using a light meter capable of measuring the intensity of individual pixels, a reasonable approximation of this value can be determined for a given monitor, but this procedure is very tedious and such equipment very expensive. Unfortunately, monitor manufacturers do not generally make this specification available, possibly because few people are aware of it. Furthermore, this value is adjustable with the *focus* controls, which are usually located inside the monitor.

Since calculating the pixel bleed for every monitor is impractical, consider instead the reason pixel bleed exists. Imagine the monitor displaying a flat field of equal intensity pixels. If the σ value is small enough that there is no noticeable pixel bleed, it would be possible to see the grid pattern of the pixels clearly on the display surface because the intensity of the light halfway between the pixels would be almost zero. As the σ value is increased, the adjoining pixels add more light to the space between them. At the optimal level of pixel bleed, the space between equally bright pixels is close enough to the brightness of the pixels that the difference is imperceptible, resulting in the impression of an area of uniform intensity. Increasing the pixel bleed further causes the area to become even smoother, but the display looses its sharpness (Klassen, 1989). While individual monitors may not be worth measuring an optimal value is calculable and we can assume that all well adjusted monitors have this value. Klassen suggests that a value of $\sigma/a = .51 \pm .01$ is optimal, where a is the distance between pixel centers. By using $\sigma = a/2$, Equation 4.3 becomes

$$\Phi(x, y) \propto \exp\left(\frac{-2((x - x_0)^2 + (y - y_0)^2)}{a^2}\right). \tag{4.4}$$

Using (4.4), it is possible to calculate the contribution that a pixel's neighbours make to its measured luminance. Figure 4.1 shows the pixel neighbours. The first and second neighbours contribute a total of 61.5% of their intensity, and the third and fourth neighbours contribute 0.2%. Thus, the measured luminance at the center of a pixel is 161.7% of the intensity being generated by that pixel. Therefore, 38.07% of the measured luminance of a pixel is contributed by neighbouring pixels if only the immediate neighbours are considered, and 38.13% if the second set of neighbours are considered. Given the significant contribution of the first and second neighbours, to calculate

[2]This is actually a simplification of the formula for a monochrome display. For a colour display, the formula is similar, but complicated by the existence of a shadow mask.

[3]The beam shape is usually not the same in the vertical and horizontal directions, and thus the value of σ should be different for the x and y directions. This difference is not significant in this context and is ignored for simplicity.

the intensity of a pixel the luminances of its first and second neighbours must be known, but third and fourth neighbours can be ignored.

Unfortunately, calculating the intensity of all pixels in a window is not a solution. It is computationally prohibitive and the accuracy of such calculations exceeds what can be used, since contrast is a single number that measures the performance of the whole screen. Therefore, the problem should be approached from the other end by first determining the average number of foreground pixels near a background pixel. The measured luminance of the average background pixel can be calculated as

$$L'_{bg} = L_{bg} + \sigma(L_{bg}N_{bg} + L_{fg}(1 - N_{bg})) \tag{4.5}$$

where N_{bg} represents the average number of background pixels neighbouring a background pixel. The optimal σ value of 0.62 would typically be used. The measured luminance of the average foreground pixel can be calculated similarly

$$L'_{fg} = L_{fg} + \sigma(L_{bg}(1 + N_{fg}) + L_{fg}N_{fg}) \tag{4.6}$$

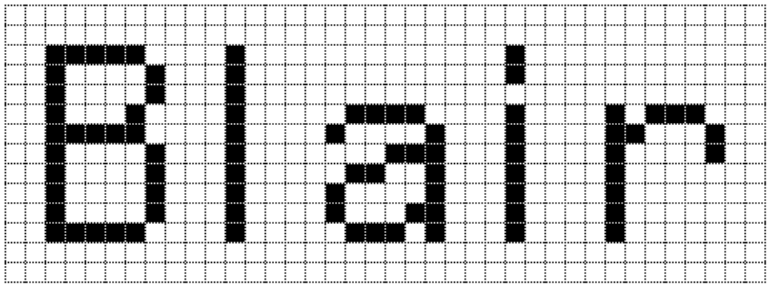
where N_{fg} represents the average number of foreground pixels neighbouring a foreground pixel. Two important observations should be made. First, N_{fg} is not necessarily equal to N_{bg}. Second, when calculating the average adjacency only pixels that are at a luminance edge should be considered because they are the ones that determine the contrast.

For text based applications, two observations can be made. First, the average neighbour of a foreground pixel, and thus the intensity of the pixel, is largely determined by the font being used. Second, the average neighbour of a background pixel is largely independent of the font being used. Consider the two fonts shown in Figure 4.2. To see how this affects contrast, first consider what happens when these fonts are rendered in black text on a white background, just as they appear in the Figure. The white background pixels will bleed into any adjoining black foreground, increasing their brightness and therefore lowering the contrast. The average foreground pixel of font (a) (the thinner font) has approximately six background neighbours, whereas the average foreground pixel in font (b) (the thicker font) as only three. As a result, the foreground pixels of font (a) have roughly twice as much light added to them as the foreground pixels of font (b), resulting in a lower contrast for font (a) than for font (b).

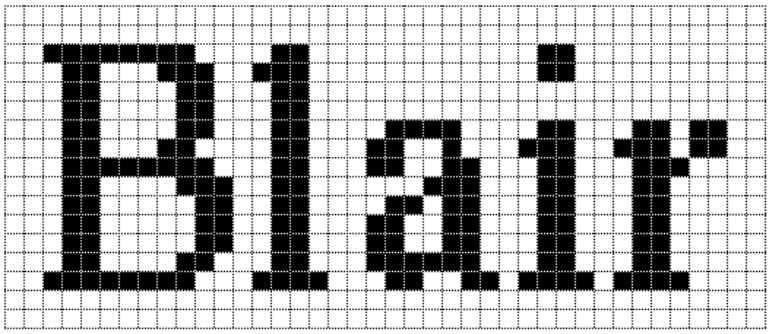
Now consider what happens when these fonts are rendered in white text on a black background. The white foreground pixels will bleed into the adjoining black background pixels, but the overall affect will be the same for both fonts. Additionally, the foreground pixels will bleed into other foreground pixels, increasing their brightness. The average foreground pixel of font (a) has two foreground neighbours, whereas the average foreground pixel of font (b) has approximately five. As a result, the foreground pixels of font (a) have roughly half as much light added to them as the foreground pixels of font (b), resulting in a lower contrast for font (a) than for font (b).

The effect in both cases is the same. Font (a) has less contrast than font (b) when displayed with the same colours. While there are many fonts available for use in most window systems, it would be possible to calculate an average foreground/background pixel adjacency relationship for each of these fonts. However, for a system with scalable fonts this is not possible. Instead, fonts can be divided into a few categories such as *thick* and *thin*. Thick fonts are those whose foreground pixels are adjacent to many other foreground pixels, similar to font (b) in Figure 4.2. Font (a), on the other hand, is representative of thin fonts; most of the pixels adjacent to its foreground pixels are background pixels. Of course, scalable fonts would be classified differently depending on the scale.

In a graphics based application, the interaction of the foreground and background is much harder to determine without significant work because the relatively simple pattern found in text based applications is not likely to exist. The best approach when no pattern can be found is to use the worst case situation, in which eight pixels of the lighter colour surround each pixel of the darker colour, significantly lightening the darker colour. While this situation may be extremely unlikely, depending on the application, this assumption provides a lower bound that will guarantee legibility.



(a)



(b)

Figure 4.2: The average number of foreground colour pixels around any pixel in a font varys widely between fonts. Over 70% of the pixels in font (a) have one or two neighbours. Conversely, well over half of the pixels in font (b) have five or more neighbours and only a dozen have two or less. In both cases, most background pixels that are neighbouring foreground pixels have 2 or 3 foreground neighbours.

Pixel Adjacency	Colour Orientation	Black Level 1%	2.5%	5%
Thick Font	Black on White	0.714	0.695	0.666
	White on Black	0.709	0.687	0.653
Thin Font	Black on White	0.521	0.509	0.490
	White on Black	0.706	0.680	0.640
Worst Case		0.476	0.465	0.449
Best Case		0.828	0.794	0.743

Table 4.2: The effects of different black levels and pixel adjacency relationships are apparent. Changing the black level is insignificant compared to the effect of changing the pixel adjacency relationship. The thick and thin fonts are those shown in Figure 4.2. The contrast was calculated using the values suggested in the text. The worst case is when a single black pixel is completely surrounded by white. The best case is when a single white pixel is surrounded by black.

Table 4.2 shows the contrast calculated for black and white foregrounds and backgrounds with an optimal value of sigma. As can be seen, the effects of using different black levels is not that significant compared to using different pixel adjacency values.

4.4 Colour Contrast Metric

By using the average adjacency relationship and the optimal σ values, a reasonable metric for measuring colour contrast can be determined by setting L_{max} and L_{min} in Equation 2.15 to the appropriate one of L'_{bg} and L'_{fg} from Equations 4.5 and 4.6, as follows

$$L_{max} = \max\left(L'_{bg}, L'_{fg}\right) \tag{4.7}$$

$$L_{min} = \min\left(L'_{bg}, L'_{fg}\right) \tag{4.8}$$

If none of the specifications of the monitor or the viewing area are provided, the reasonable defaults provided throughout this chapter can be used. Chromaticities for the phosphors of a standard, or similar, monitor can be used to calculate the *Y* values needed for Equations 4.1 and 4.2. A typical black level such as 1% can also be used in these equations. The background lighting and the contrast and brightness controls on the monitor can be ignored. Optimal pixel bleed can be assumed for the σ in Equation 4.4. Using these default values, Equations 4.7 and 4.8 can be used with Equation 2.15 as a reasonable metric for calculating contrast on a CRT.

Chapter 5

Colour Constraints

This chapter discusses the issues involved with adding colour constraints to a dynamic window system with the goal of assisting with aesthetic colour selection. It should be kept in mind that the goal of this thesis is to show that this is possible, not to provide a definitive solution. Indeed, an important aspect of the current implementation is to experiment with such issues.

Before tackling the issues of adding colour constraints to a window system, dynamic window systems are described generally. Following that, general properties of colour constraints are discussed. Next, techniques for avoiding the disruption of user colour associations are suggested. The bulk of the chapter investigates specific ways colour constraints can assist with aesthetic colour selection. Finally, the features of dynamic colour constraints that allow them to be successfully added to a dynamic window system are summarized.

While this chapter discusses issues on a general level, the reader should keep in mind that this design has been implemented. The implementation is discussed in Chapter 6.

5.1 Dynamic Window Systems

Consider how a user interacts with a window system, which was briefly alluded to in Section 1.5. Most current window systems are based on the assumption that no arbitrary restrictions should be placed on the user. In other words, the user should have complete control of the windowing environment whenever possible. However, most systems fail to realize that users do not always want or need absolute control. Some control is essential, but beyond that the user performs tasks that could be done equally well by the computer.

Thus, that while users should be *allowed* to specify anything, there should be tools available to make specifications the user does not, or can not, provide. Most window systems, lacking such tools, *force* the user to specify everything. The result is a failure to achieve the original goal. To summarize, the typical window system is designed to allow the user complete freedom. But that freedom is consumed by requiring the user to perform tedious and complicated tasks that are actually unnecessary.

There is another problem when the user specifies everything with little help from the window system. By their very nature, windowing environments are dynamic, constantly changing as the user interacts with them. Windows open and close, are moved and resized. While users typically have an intuitive idea of how they want the environment to appear and be organized, they typically have no way of communicating it to the window system. As a result, common actions such as opening or closing a window often require other windows to be adjusted, and the user must perform these adjustments.

It is better to tell the window system not only *what* to do, but *why* it is being done. *Why* can be expressed in the form of *constraints*. For example, the user may position the windows so that a particular window is visible. By creating a constraint that this window should be visible, new windows can automatically be positioned to keep it visible. In addition, it is possible to specify the *why* without specifying the *what*. For example, if the system is informed of the visibility constraint before the window is opened it can automatically position the window so it is visible. In either situation, adjustment of many windows by the system may be needed to satisfy the constraint.

The user is not necessarily relinquishing any freedom by allowing the system to help in managing the environment. Currently, users must tell the system to place a window at an exact position, which can be equally well expressed as a constraint. However, the crippling burden of being *forced* to specify every detail is lifted from the user's shoulders, freeing them to concentrate on their work.

Traditional window systems are normally *static* and lifeless in their interaction with the user, doing exactly what the user tells them; no more, no less. By contrast, the window system described here is more suited to the *dynamic* nature of windowing environments, interacting with the user's actions and filling in details. Therefore, such a window system shall be referred to as a *dynamic window system*, in contrast with the more common *static window system*.

There are a few important requirements that should be satisfied by a dynamic window system if it is to be useful:

- *Superior results.* Inferences made by the window system should never result in a situation that is less desirable than what would have occurred had no inferences been made. For example, many current window systems default to a black and white colour scheme which, while bland, satisfies the basic functional and aesthetic requirements. A system that attempted to select more exciting colours, but that frequently created nonfunctional or offensive combinations would not be acceptable.

- *Non restrictive.* User specifications should take precedence over selections inferred by the window system. If the user wants to select colours that exhibit undersirable properties or to position windows in a potentially ambiguous arrangement, they should be allowed to.

- *Easily customizable.* It should be very easy for the user to change anything that is inferred by the system, or to alter the method used by the system to perform its inferences.

- *Predictable.* While it is desirable that the window system assist the user with tedious tasks, the system should be predictable so that the user feels in control. If the systems appears to move windows or change window colours for no apparent reason, the user may become disoriented or annoyed with the system. To make the system predictable, changes should only occur in response to user actions. Equally important, the changes must start immediately so that it is obvious which action initiated the change. For example, if a new window is opened and no changes occur until a few seconds later, the user may not realize that the opening of the window initiated the change, lessening the predictability of the system.

- *Adequate Performance.* While perhaps obvious, the performance of the window system must be considered because dynamic window systems must perform significantly more work that static window systems. Maintaining constraints should not noticeably degrade the response time of the window system. The constraint solver is one of the lowest priority tasks that need to be performed by the window system and should not interfere with more important tasks such as sending keyboard and mouse input to applications and displaying the results of these actions.

Schlueter (1990) represents a significant step in the direction of creating more dynamic window systems. Criticisms of the system result from violations of one or more of the above requirements. For example, one of the properties of the system is that overlapping or adjoining windows cannot have coincident text baselines, which

is beneficial in general. However, occasionally a user wants text baselines to be coincident, such as when the contents of two windows are being compared. The system provided no way to do this, a violation of the "non restrictive" requirement.

Two important lessons can be learned from Schlueter (1990). First, it is possible to create a dynamic window system. Most of the problems with the system have obvious solutions that could be implemented in a production system. Second, to create a system that is usable, these issues must be addressed. Otherwise, the system is likely to exhibit annoying behaviour which will decrease its user-acceptance. Having said that, it should now be recalled that the aim of this thesis is not to create a commercially viable product. Many features that are needed in a commercial product, such as an intuitive graphical interface, are beyond the scope of this thesis. The focus of this chapter is on the constraints are needed to satisfy particular requirements of the system; how the user generates these constraints is not addressed.

5.2 Dynamic Colour Constraints

How can a dynamic window system with colour constraints assist with aesthetic colour selection? Consider some examples of colour usage from Chapter 3. They can be expressed as constraints on colour selection. For example:

- Red is often used to represent danger or warning, an absolute property of the colour red, created by societal influences on the typical viewer. However, when an application designer uses red to signify danger, their desire can be interpreted as a request for a colour value from a range of values that appear as red.
- An important relative functional requirement is the maintenance of enough contrast between background and foreground colours for legibility, which can be expressed as the requirement: the contrast between foreground and background must exceed a certain level.
- Another common example occurs when colour is used to show the relationships between windows. This can be expressed as a requirement that unrelated windows use significantly different colours and similar windows use similar colours.

Like these examples, other issues can be viewed as constraints. There are some very general observations that can be made about the sort of colour constraints that are of interest in this thesis.

5.2.1 Multiple Constraints

Few colours can be entirely described by a single category of colour usage, as discussed in Section 3.1.3. Similarly, few colours are affected by only one constraint. Each constraint is simple and predictable, but the complete set is necessary to describe most colours.

5.2.2 The Categorical Division of Colour Constraints

The distinction between absolute and relative colour usage divides constraints into two groups distinguished by the number of colours involved in the constraint. Constraints derived from absolute colour properties do not refer directly to other colours and can be expressed as a function of a single colour. Relative constraints, on the other hand, refer to two colours and must be expressed as a function of both colours.

The distinction between functional and aesthetic colour usage creates another division among the constraints. The objective nature of functional constraints means they can generally be expressed algorithmically. Aesthetic

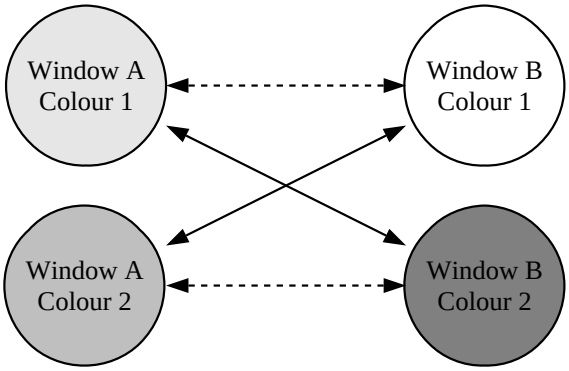


Figure 5.1: High level constraints can be expressed in a hierarchical fashion allowing complicated constraints to be decomposed into more simple ones. This hierarchy expresses a constraint that ensures two windows are different from each other, with the colours represented as circles and the constraints as arrows.

constraints, on the other hand, are thought to be more subjective which precludes easy algorithmic expression. This difference accords well with the goals of this thesis. When constraints to solve functional usage problems are embedded in the window system, the user is freer to concentrate on the aesthetic issues.

5.2.3 The Hierarchical Nature of Colour Constraints

Colour constraints are often expressed in a hierarchical manner. Consider the functional constraint that unrelated windows should be easily discriminable. Given two unrelated windows, this relationship can be expressed hierarchically, as shown in Figure 5.1. The two background colours in each of the windows must be different from both of the background colours in the other window, with the degree of difference shown by the weight of the lines, with heavier lines implying a need for further separation.

5.2.4 The Varied Importance of Colour Constraints

Not all of the constraints expressed by the system are of equal importance. For example, the constraint that window contents be legible is far more important than any other constraint. The window system therefore allows constraints of high importance to be satisfied in preference to constraints of lesser importance.

5.2.5 The Dynamic Nature of Colour Constraints

Any colour choice is determined by a (possibly complicated) set of inter- and intra-window constraints. When a window is created or destroyed these constraints, or the solution to them, may change. In a dynamic window system the constraints are re-evaluated whenever the window configuration changes. The system discovers when current colour values are unacceptable and changes them to satisfy the constraints. The automatic changing of

colours has significant implications, the most important of which is the effect it has on colour association, which is discussed in Section 5.3.

Another interesting issue is user acceptance. In the dynamic window system created by Schlueter, the reaction to, and acceptance of, the system was surprising. At first users were disturbed when windows moved of their own accord since unexpected motion is inherently distracting. After using the system for a short while, however, users became accustomed to the behaviour and began to take advantage it. Most surprising, however, was the reaction of these users when they returned to using more traditional window systems. They reported that the systems felt dead and lifeless and were disturbed by the lack of motion! While these tests where informal, they suggest that a properly designed dynamic system is acceptable to users.

It is likely that user response to a dynamic window system with colour constraints will be the same or better because tools for colour selection are significantly poorer than those available for window positioning. Therefore, users have fewer preconceived notions of how colour should behave.

5.3 Potential Problems With Colour Association

Window colours in a dynamic window system with colour constraints occasionally change without explicit direction from the user, which raises the possibility of potentially harmful effects on colour associations.

Christ (1975) reviewed the experimental literature and found that colour is superior to size, brightness, and shape in searching for and identifying items that vary in only one of these categories, making colour ideal for organizing windows. In addition, for some tasks people remember colour longer than size, orientation or shape so that colour is an excellent way of presenting context.

These functional benefits of colour depend on the user's ability to associate a colour value with some semantic meaning. Colour helps in context resolution, for example, because the user associates the colour of the window with the task for which the window is being used.

Unfortunately, changing the window colours might destroy many of the potential benefits of colour association. Worse, dynamic colour might be less beneficial than even arbitrary colours in a static window system, since colour associations are static, regardless of aesthetics. To prevent this from occuring, one or more of the following strategies can be considered.

1. To prevent loss of context, colours should be changed gradually instead of immediately adopting their new colour values, so that the user can adapt to them. Gradual changes can be accomplished either by making only small changes or by slowly changing colours from one value to another. For example, assume a new window opens which must have colour values that are a well distinguishable from those of current windows, and this is not possible without changing the colours of the other windows. If the colours immediately change to their new values, the changes destroy any associations the user had developed with those windows. However, if the window colours changed slowly, the user can notice the changes and adjust to them. The colours should also change slowly for aesthetic reasons. If a large number of colours suddenly change, the effect is unpleasant and visually jarring. However, by slowly changing the colour values, the effect will be smoother and, hopefully, more pleasing.

2. Any individual colour value should change as little as possible. For example, if a colour changes from one shade of purple to another shade of purple, loss of association will be less likely than if the colour changes to some dramatically different colour such as blue or green.

3. To give persistence to colour values, older windows should change less than newer ones. This is important, because experience shows that associations may be stronger with older windows. In particular, when colour

changes are the result of a new window being created, the new window's colours should be changed in preference to any other window's colours, since there will be little or no contextual associations with the new window. If colour constraints can be solved by only changing the new window, or by only changing existing windows slightly, the user's colour associations are disturbed as little as possible.

4. Finally, colour associations last not only during one session with a window system. After using the same colours for a relatively short period of time, the associations of colours with tasks become strong enough that they extend between sessions. Thus, for colour associations to be maximally preserved, the state of the colour in the system should be remembered between sessions. The implementation is bound to be highly system dependent. Some computer, like the MacIntosh, already preserve the state of the system between sessions. Folders that were open when the machine is shut off will be re-opened when it is turned back on. Other systems, such as most window systems for machines running the Unix operating system, have a poorer concept of inter-session state so adding this feature will be more difficult. Cowan and Wein (1990) examine the differences between state and history based interfaces, as well as the pitfalls of adding state information to a history based window system. A major pitfall to remembering colour between sessions with a history based window system is that the user may incorrectly infer that the entire state has been remembered. Therefore, if colour is remembered, some thought must be given to preserving as much of the state as possible.

5.4 Assistance for Aesthetic Colour Selection

Aesthetic colour selection for window systems is as a two step process. First, the general characteristics of the window's appearance and the relationships between the colours, the *window style*, are chosen. The window style can be simple and abstract, such as deciding that the window borders should be darker than the application area in the center. It can be complicated, such as wanting the borders to be a light pastel colour, the center area to be a slightly lighter analogous colour, the text in both areas to be dark saturated colours and to have all of these colours harmonize. It can be detailed, such as defining the exact values for all of the window colours. Second, colour values are chosen to satisfy these characteristics and any other constraints on the window colours, especially functional constraints relating to contrast and window organization.

Currently, there is no way provided for the user to specify relationships between colours, so only the most trivial of window styles can be specified.

The preceding sections showed how colour relationships may be expressed as constraints. However, to use constraints effectively for aesthetic colour selection, semantic information must be attached to individual colours on the display. Examples of semantic information include whether a colour is a background or foreground colour, if it is used in the border or application area of a window, what application created the window and what task the application is being used for. Some window systems, such as X11, already have semantic information attached to colours but make very little use of it (see Section 6.1).

By attaching semantic information to colours, it is possible to add colour constraints to a dynamic window system. These constraints can assist with aesthetic colour selection in the following ways:

- Functional constraints can be applied automatically.
- Possible aesthetic colour combinations can be suggested.
- More abstract colour specifications are possible.
- Useful defaults can be provided.

- The system can be customized gradually.

In the following sections, each of these is discussed to show how it can assist with aesthetic colour selection and how it can be provided using constraints.

5.4.1 Automatic Handling of Functional Constraints

To allow the user to concentrate on making aesthetic colour choices, the window system should automatically handle some of the functional constraints. The two most common functional constraints, *contrast* and *window organization*, are described below.

5.4.1.1 Contrast

In Chapter 4 the importance and difficulty in calculating contrast is discussed and a reasonable metric for calculating contrast on CRTs is presented. Using this metric, it is straightforward to create a constraint that enforces contrast by ensuring there is sufficient luminance difference between any pair of foreground and background colours. As discussed in Chapter 4, if the context in which the colours will be used is known, the contrast can be determined more accurately. Otherwise, the worst case suggested in Section 4.3 can be used. Fortunately, it is relatively easy for an application to inform the window system if the foreground colour will be used for a specific purpose such as displaying text using a certain font.

As was pointed out in Section 4.1, while the absolute minimum contrast threshold is approximately 12%, a much higher threshold of as great as 80% of the maximum available contrast may be needed to ensure comfortable reading. Additionally, some users will want to adjust this threshold to account for poor vision or aesthetic preferences, so the threshold used in this constraint should be user adjustable.

5.4.1.2 Window Organization

To create constraints that assist with window organization, the system must know the relationships between windows and have a metric to determine if windows are visually similar or visually different. Using this information, constraints that group related windows and create visual distinctiveness between unrelated windows can be created.

Window Relationships. There are many ways the system could be informed of the relationships between windows. A possible approach is to have the user specify them, which is inconsistent with reducing the workload on the user. Instead, the system should infer reasonable default relationships. Of course, the user should be able to change the defaults or specify additional relationships.

A simple technique for inferring relationships between windows is to derive them from the semantic attributes associated with the windows. Semantically similar windows should be coloured similarly, windows that are semantically different should be coloured differently. Some semantic attributes, such as the name of the application program that owns the window and the machine the application is running on, can be automatically determined by the system. Additional semantic attributes, such as the type of work being done in the window, can be specified by user. However, some groupings require more than one of these semantic attributes. For example, users may wish to group windows according to all three attributes mentioned above or to use other attributes, such as the type of file being editing in an editor window. Therefore, there should be no restriction on the type of attributes that can be attached to a window, and the method used to determine which windows are “different” and “similar” should be adjustable by the user.

Window Difference. A general model of how windows can be judged as related or unrelated has not been developed and doing so is beyond the scope of this thesis. However, since a method for judging the visual difference and similarity of windows is needed, the following simple scheme is proposed. First, notice that the dominant colours in any window are the background colours of both the applications area and the border. For example, if the background colours of two windows are the same, some relationship will be assumed to exist between these windows. If the background colours are different, no relationship will be assumed, regardless of the foreground colour values. There are exceptions, of course. If all the background colours are very desaturated and dull, and all the foreground colours are extremely saturated, then relationships might be drawn based on the foreground colours. In general, however, foreground colours can be safely ignored. Given this, windows can be judged to be different if the backgrounds are different, and the same if the background colours are the same. To implement this constraint, it is necessary only to measure the difference of the background colours, as illustrated in Figure 5.1.

Colour Difference. Most definitions of "different" and "similar" windows, such as the one presented in the previous section, depend on judging the difference of one or more colour values. Recent experimentation by Boynton and Smallman (Section 2.6) shows that the ability to segregate colours is dependent on their separation in a uniform colour space. They found that basic colours segregated well because they are well separated in the OSA colour space, with an average interpoint Euclidean distance of greater than ten OSA units. They also found that in special cases, distances of only five units provided sufficient segregation. They did not extend their study to find a lower bound on the distance that provides good segregation, but their work can provide a good metric. Table 2.3 shows that a distance of ten OSA units can be used as a threshold for good colour separation. However, this distance is only an estimate based on the separation of the basic colours, and the measurement of difference in any uniform colour space, including the OSA space, is not exact. Thus, the constraint that enforces colour difference does not use this value as a fixed threshold. Rather, it rates the difference between the colours as progressively worse the farther under ten OSA units it falls. Therefore, if it is possible for two colours to be separated by the threshold value they will be, but if other constraints draw them closer together, the penalty for a decreased distance will be insignificant for values close to ten, but gradually become more significant as the distance decreases.

Measuring Colour Distance. Boynton's work was done in the OSA colour space which is not particularly useful when working with a computer display, as it is neither continuous nor does it apply to the entire gamut of colours available on a CRT. However, colour difference works well in the OSA space because it is perceptually uniform. Thus, other perceptually uniform colour spaces can serve equally well, such as the CIELUV colour space. Appendix A shows that the work done using OSA interpoint distances can be applied reasonable well using CIELUV interpoint distances. The advantage of CIELUV is that an easy conversion exists between it and colour spaces commonly used in computer graphics, such as RGB and HLS. Appendix A shows that ten OSA units is approximately equal to eighty CIELUV units, which can be used as the threshold for the distance constraint for different windows.

5.4.2 Suggest Colour Combinations

Section 2.8 discusses the difficulty in producing a general model of colour harmony. The colour choices people make in their everyday lives show that colour preferences and colour harmonies are very subjective. Thus, expecting a computer window system to select harmonious colours for windows without user assistance is unreasonable.

However, expecting users to make harmonious colour choices may be equally unreasonable. While it is quite easy for most users to say whether they like a particular colour combination or not, it is significantly harder for

them to actually generate colour combinations they find appealing. Creating harmonious colour combinations is something that artists and designers take years to learn, and most users of window systems are not trained artists or designers. Typical users make far better critics than designers.

Therefore, a natural way to assist users in making aesthetic colour choices is to suggest various colour combinations and window styles, and let them criticize. Consider the task of selecting individual colours via suggestion and criticism by providing the user with a small palette of colours from which they can select a single colour value. This approach is becoming more common, being used in commercial systems such as the Property Manager for OpenWindows2.0. However, these systems do not take the semantics of the colours into account when presenting palettes of colour values to the user, and are therefore of limited value.

The problem of suggesting colour combinations and window styles for individual windows is more difficult. Unlike the situation where a single colour is being suggested, colour harmony is a prime concern. Although it is impossible to create a general model of colour harmony, it is possible to generate more simplistic models with more narrow applicability. Consider the colour schemes suggested by Quiller (Section 2.4). They all have the property that one or two colours are selected, and the remaining colours are generated in relation to those colours. This approach can be applied quite naturally to windows. Given a dominant colour value and one of Quillers simple colour schemes, an instance of the scheme can be selected which contains the dominant colour value. The range of colour values that fall within the specific instance of the colour scheme can then be calculated. Creating a constraint that restricts colours to have values within this range is straightforward.

Thus, the user selects one colour as the dominant colour and the constraint is applied automatically to select the remaining colours. For example, a monochromatic scheme can be created by selecting any colour hue for the dominant colour and constraining the remaining window colours to use the same hue. Similarly, an analogous scheme can be created by selecting one of the possible analogous colours schemes that contains the dominant colour value and constraining the remaining window colours to use an analogous hue. The complementary and split complementary colour schemes can be defined in similar ways. It should be noted that these colour schemes restrict only the hue of the colour values. When Quiller creates a colour scheme, he includes all the "semineutral" colours that fall between the fully saturated hues and neutral grey. When specifying colours using the HLS model, this specification allows any saturation. Similarly, any colour values can be mixed with white or black. Adding white to a colour is roughly equivalent to increasing the lightness above the midway value, adding black to a colour is roughly equivalent to decreasing the lightness below the midway value. In both cases, the colour value is also desaturated. Quiller's colour schemes are based on subtractive colour mixing, whereas HLS is additive, so the complementary colours in HLS are different from the ones Quiller uses. How this affects the colour harmonies is a matter for future work. In any event, the natural correspondence to the method Quiller uses to mix colours is a good reason for using the HLS model for colour specification, as opposed to HSV or RGB.

Each colour can be generated to fall randomly within the colour scheme, subject to whatever other constraints act on it. With the possible exception of the monochromatic scheme, each of these schemes contains a wide variety of colours that can be combined to create a remarkable array of window styles. More refined window styles can be created by applying additional constraints to limit the possible saturation and lightness values. For example, if the border colour was constrained to be very dark and the background very light, a distinctive monochromatic window style is achieved. However, by using more unconstrained colour generation schemes initially, users are prompted with a greater variety of window styles and colour combinations, many of which would never have occurred to them, and can learn over time which styles and colours they find attractive. They can then apply additional constraints to restrict the possible the window styles and colour combinations to those that they find most attractive.

Because they are prompted with different colour combinations and window styles, even users untrained in art or design gradually become familiar with combinations of colours and window styles that appeal to them. The window system *trains* the users sense of colour harmony through constant experimentation, in much the same

way artists and designers are trained. As the user develops a sense of colour harmony, he or she can develop more elaborate colour styles that more accurately express his or her personal taste.

This approach of having the window system select colours randomly from a restricted set is workable as long as the following conditions are met:

- it must be very easy for the user to have the system pick another set of colours. If the selected colour combination is entirely unsatisfactory and the user does not want to change it manually, he or she should be able to tell the system to try another scheme.
- it must be very easy for the user to modify the individual colours that are generated by the scheme. If a colour combination is selected by the window system and the user wants to change one or more of the colours to create a combination he or she prefers, it should be possible.

In addition to these necessary conditions, one or more of the following conditions will further improve the usability of the system:

- one or more of the colours can be frozen and the system told to pick a new set that incorporates them.
- users can inform the system which aspects of the colour set they like or dislike and the system can use this information when making future selections.

5.4.3 Abstract Colour Specification

Abstract colour specification refers to any colour specification that tells the system how to pick a colour instead of telling it the exact colour value to use. Examples of abstract colour specifications are found in all four of the categories of colour usage. Many of the constraints discussed in other sections of this chapter are also examples of abstract colour specification.

The constraints discussed in Section 5.4.1 are abstract colour specifications which can be categorized as relative functional colour usage. For example, a useful foreground colour could be completely specified by the constraint that there should be adequate contrast between the foreground and the background. Unless other constraints are imposed to further restrict the colour there is no guarantee that the result would be visually appealing, but it would at least provide adequate contrast.

Many absolute functional colour concepts are more accurately expressed in an abstract manner than as specific colour values. Consider, for example, the use of red to represent danger or as a warning. In most cultures this property of the colour red is absolute. However, an application designer who wishes to use red to signify danger might think "I wish to use a colour that represents danger, so a colour that the viewer interprets as red would be appropriate. Therefore, the colour should be constrained to appear to all viewers as red." However, there are many different colours that appear as "red", all of which would serve the purpose of signifying danger equally well. Therefore, specifying a specific value of red is not appropriate. To see how a more abstract specification for "red" can be formulated, recall Boynton's study of basic colour terms discussed in Section 2.6. Boynton and Olson identified sets of colours in the OSA colour space that correspond to the basic colours (Boynton and Olson, 1987). Using this data, a constraint that restricts a colour value to "red" could easily be constructed by restricting the colour to a colour value that is in Boynton and Olson's set of "red" colours.

The concept of window styles introduced in Section 5.4.2 is an example of abstract colour specification that can be categorized as relative aesthetic colour usage. For example, a user may wish to have green terminal windows and blue editor windows, but aside from the basic hue does not care about the specific colour values used for any of the window colours. To allow the user to specify colours in this fashion, a set of window styles can be provided

using the simple colour schemes discussed in Section 5.4.2. Additionally, by using the default window style they only need to specify "I want the major colour of my terminal windows to be green" and the system will select the specific colour values.

Finally, some absolute aesthetic colour choices are more appropriately expressed abstractly. Consider the above example of a user specifying that terminal windows should be green. The specification of "green" is an absolute aesthetic colour choice. However, the user may not wish to specify the exact shade of green, such as olive green, kelly green, forest green, etc. In this case, the colour is better defined by a constraint which ensures that its colour value is identifiable as "green," just as the absolute functional colour "red" was defined above. In this case, the same approach can be taken by using Boynton and Olson's quantification of the basic colours. Many abstract specifications of absolute aesthetic colour usage can be expressed as some function of the basic colours because of the nature of the basic colours.

5.4.4 Reasonable Defaults

An important function that is performed by a window system is choosing default colours for the windows. The standard approach is to choose a simple colour set, such as black text on a white background and a subdued border colour with black border text, and use these colours for all windows. This approach is unsatisfactory for the following reasons. First, the ability of colour to organize windows is lost. More than one user has commented that they didn't realize that colour could be used to organize their windows. Part of the blame for this ignorance lies with the window system for not providing examples of how colour can be used. Second, the window system appears utilitarian and visually boring. The complete lack of colour variety does not entice the user to create more interesting colours, rather it reenforces the belief that computing environments are cold and impersonal. By using a utilitarian colour scheme, users may also get the impression that changing the colour values is difficult and avoid attempting to do so because they do not want to waste a large amount of time. While this feeling is quite justified in many current window systems, it need not be so.

The system defaults should be pleasant and colourful. They should whet the user's appetite for more exciting and novel colour schemes. However, they should also be inoffensive to the vast majority of people and satisfy some basic functional constraints, such as those discussed in Section 5.4.1.

To achieve defaults that satisfy these requirements, the following approach is used. A window style is designed which, given a single colour, generates the remaining window colours using a simple and bland colour scheme. A good colour scheme for this purpose is the monochromatic colour scheme discussed in Section 5.4.2. By using standard design principles, such as those summarized in (Meier, 1987), a window style such as the following could be created (the specifications in this example use the HLS colour space as suggested in Section 5.4.2):

- *border colour.* Any hue, a saturation of 25%, a lightness of 50%.
- *border text.* Absolute colour value of black.
- *window interior.* Same hue and saturation as the border, a lightness of 90%.
- *window text.* Absolute colour value of black.

This window style generates bland colours that satisfy the requirements for a default window style. In addition, the contrast in this example scheme is very high, ensuring that legibility is satisfactory. Finally, the undesirable perceptual phenomena discussed in Section 2.5 are avoided by using a monochromatic colour scheme since only a single hue appears in each window.

However, this window style does not satisfy all of the requirements stated above. The functional requirement that colour be used to organize windows also exists. The eleven basic colours provide a method to satisfy this

goal. First, black or white should be used as foreground colours so that, as discussed above, the basic requirement of contrast is satisfied and additional perceptual problems are avoided. Grey is the best choice for the overall background colour of the window system, since no colour will conflict with it when used as a border colour. As discussed by Boynton, the eight remaining basic colours are excellent choices for window colours when the primary motivation is window organization. In addition to their good segregation qualities, they allow windows to be easily and unambiguously referred to by colour name. Given this set of colours, the default window organization discussed in Section 5.4.1.2 is used to decide which windows should be grouped together. The application name and a user supplied name are used to provide a default organization for windows. If the user does not specify a name, the machine the application is running on is used as the default name of the window. The latter point is especially useful in an environment where applications are often run on different machines. When a new window is opened, if it is to be grouped with an existing window it uses the same basic colour for its dominant colour as is used in the existing window. Otherwise, an unused basic colour is assigned to it. If more than eight different colours are required, Boynton and Smallman point out that colours midway between the basic colours can be used.

5.4.5 Allow Gradual Customization

An extension of the inability of current window systems to specify colour relationships is that colour customizations tends to be all-or-nothing. When the user decides to add colour to their environment, they must specify very many colours. The window system does not force them to change all the colours, of course, but the interrelationships of the colours requires that many colours be specified for real benefit to be gained.

Consider the following example. A typical window system defaults to a single colour scheme for all windows, such as black and white, or some colour and either black or white. The first customization many users perform is to change these global defaults, which is painless, requiring only two or three colours to be specified. However, at some point many users realize that colour can be used to help them organize their windows. In order to use colour to organize the windows, however, many colours need to be specified because each window must have its colours explicitly defined.

Unlike conventional window systems, the defaults used by the dynamic colour system automatically use colour to organize the windows. Therefore, the customizations most users want to perform fall in two categories. First, the default aesthetic choices made by the window system may need to be adjusted. Second, the user may wish to change the method by which the window system organizes the windows.

5.4.5.1 Aesthetic Customizations

There are many aesthetic customizations that can be performed. Some new constraints and modifications of existing constraints are obvious, others are more difficult. The more common customizations are discussed below.

Expressing Dislike of Colours. The first customization the vast majority of users perform is to avoid colours or colour combinations they dislike. The window system chooses colours for unrelated windows from the basic colours: red, green, yellow, blue, brown, purple, pink and orange. Few people find all of these colours equally pleasant. Occasionally, the window system bases a colour scheme on a basic colour that the user dislikes. It should be very easy for the user to tell the system not to use that colour again. Given this information, complying with the user's wish involves removing the offensive colour from the set of colours from which the system chooses its defaults.

Consider instead the situation where a user indicates that a particular shade of a colour is offensive. While it is quite easy to add a constraint to the system that repels colours from the immediate area around the indicated colour, other questions must be answered in order to determine the area to be avoided. Is the colour at the center

of the range of undesirable colours? How large an area around the colour should be avoided? Is the particular colour value unappealing in general, or only when used for the particular part of the window? If the colour value is unappealing only in the context of the current colour scheme, perhaps the colour scheme should be criticized, not the colour value, as discussed below. Once the range of colour values that should be excluded has been determined, a constraint can be added to the system which repels all colours from this range of values. It should be possible to apply this constraint to either the particular colour scheme or to the entire system. More often than not the constraint will be applied globally, as users see colours they dislike and want them to not occur in any window.

The second aspect of the problem, when the system has chosen a colour combination that is unattractive, is more difficult to rectify. Selecting another set of colours within the bounds of the active constraints is trivial; simply generate another random set of colours that satisfy the constraints. The problem is that the system should avoid reselecting the unattractive set of colours. Like the situation where the user is criticizing a particular colour value, it should be possible to specify a constraint that disallows a combination of colours globally or within a particular window style. It is more appropriate for these constraints to be applied to the window style, as opposed to globally. A window style represents a way for windows to be coloured and therefore criticizing a particular colour set is actually a method of customizing that window style. Creating a hierarchy of constraints that causes the a window to avoid a particular colour combination is straightforward; a constraint is created for each colour in the window which repels it from the appropriate colour in the undesirable colour scheme.

The problem with both of the above constraints is that as the set of undesirable colours and colour sets grows large the load on the constraint solver increases dramatically. It is easier for the user to tell the system that they dislike a certain colour or colour combination than to modify the window style that is generating the offending colour combinations. Indeed, it may not be possible to modify the window style without unduly restricting the colour sets that may be generated. Techniques to optimize these constraints should be investigated, but doing so is beyond the scope of this thesis.

Expressing Approval of Colours. Just as users dislike colour values or combinations of colours values, so do they approve of colour values or sets of colour values. There are two aspects to expressing approval, the first for individual colour values and the second for sets of colour values.

The fundamental difference between creating constraints that express a preference for a colour value and those that express aversion is that the constraints for colour preference cannot reasonably exist in the constraint solver on a full time basis. If they did, all instances of affected window styles would be attracted to the preferred colours. This contradicts one of the goals of the system, which is to suggest new colour combinations. Instead, the constraints should be used to alter the mechanism for picking new colours. By biasing the selection of individual colour values and the creation of colour schemes to those that the user has expressed a preference for in the past, the system will occasionally select colour schemes based on user preferences.

There are a few ways this can be accomplished. The simplest technique is to store a list of preferences and occasionally use one instead of picking colours at random. While the random colours are still unaffected by the users preferences, the occasional colour value or colour combination reflects the preferences. More complicated and powerful techniques are possible, of course. For example, user preferences could be used to create a neural network that picks colour sets similar to the ones the user expressed a preference for (Salomon and Chen, 1989). One of the criticisms of using neural nets to select colours is that they always generate sets of colours similar to the seed sets. In this case, however, that is exactly what is desired.

Modification of Window Styles. Window styles are introduced as a technique of grouping aesthetic constraints which express the general characteristics of window appearance. The default window styles are designed to create

bland colour schemes which are visually inoffensive to the majority of users. Eventually users will modify the default window styles or create their own.

A window style consists of a set of constraints which together specify how a window should appear. Each constraint is based on either an absolute or relative requirement, the absolute constraints affecting only a single colour and the relative constraints relating one colour to another. The constraints are independent of each other and can be mixed and matched relatively freely. Therefore, to modify a style, change the constraints that define it.

Exploring an intuitive interface for such window style and constraint editing is beyond the scope of this thesis, partly because an effective method of modifying the styles is dependent on their implementation. However, the constraints are fairly simple, as shown by the default window style in Section 5.4.4, so editing them should not be difficult.

Exact Colour Specification. One constraint that will often be needed in a window style is the specification of an exact colour value. To implement this, simply constrain the colour to the desired value and define the importance of the constraint high enough that the colour is not affected by other constraints. Such constraints mimic colour specification in all current window systems. Exact colour specification therefore demonstrates that colour specifications which are possible in other window systems are possible in a dynamic window system with colour constraints, so no flexibility has been sacrificed.

Manual Modification of Colour Values. When the system selects a set of colours, users may wish to modify one or more of the colours by hand to create a colour scheme that appeals to them. While this is a straightforward operation, the user's motivation for modifying the colour should be considered so that appropriate action may be taken. Usually, the user modifies colours to create an attractive colour scheme. After modifying a colour, therefore, the user may desire that the colours in that window should be fixed at their current values so that the customization is not lost. Alternatively, they may only wish that the particular colour they modified remain where they set it, leaving the other colours free to change. Finally, they may change a colour only because it does not fit with the other window colour values and do not mind if it changes when the other window colour values change. An interface for changing colours can easily ask the user which of the above situations they intend.

If the user intends to have one or more of the colours fixed at its current value, the system should not changed it. However, when the user specifies that a colour should no longer change there is usually an implied qualification that the colour can change if there is enough pressure from other constraints. In this case, constraints may be added which force the colour toward the desired value, but have greater importance so that other constraints acting on the colour will only change the value in extreme cases. Additionally, it should be possible for the user to say that they want the colour to remain exactly as it appears, in effect disabling all the other constraints acting on that colour.

5.4.5.2 Customizing Window Organization

The second aspect of customization is to modify the approach used by the system to organize windows. While the default technique of using the application name and optional user specified name is surprisingly powerful, there are times when it is not powerful enough, such as when users wish to create additional levels of grouping. For example, a user may wish to use a certain colour scheme for personal windows and another for work windows. Within the work windows, they may wish to use green for windows associated with one project and blue for windows associated with another. Within a certain project, they may want to use different colours for editing documentation than for editing source code. While it is possible to create groupings like this using the current scheme by careful creation of window styles, there should be simpler ways for the user to express these desires.

In a sense, where typical window system force the user to specify all of the colour values, the addition of constraints adds a level of indirection by allowing users to specify more abstract properties of the windows. The next step, which is needed to provide additional customization to the methods of window organization, is to create a system which will add yet another level of indirection by providing more intuitive and abstract ways of specifying the relationships between window colours. However, such a system is beyond the scope of this thesis.

5.5 The Viability of Dynamic Colour

At the end of Section 5.1 five necessary qualities of a usable dynamic window system were discussed. The features of dynamic colour that fulfill these requirements are summarized here.

Superior results. By purposely choosing bland window styles by default, the dynamic window system will create sets of colours that are inoffensive to the vast majority of users, yet are far more interesting than the defaults provided by conventional static window systems. In addition, the functional potential of the windows is obtained by selecting default colours that organize the windows in a reasonable fashion.

More importantly, by allowing gradual and abstract customizations, the window colours are more likely to remain harmonious than if the user is forced to select many specific colours to organize their windows.

Another important feature of constraint-based systems is that when all constraints cannot be satisfied, a reasonable constraint solver allows constraints to be relaxed in order to find a non-optimal minimum value. In other words, the system fails gracefully. For example, if the desired level of visual separation between windows or contrast between colours cannot be achieved, a minimum is found that provides a reasonable solution. The system does not give up when a constraint cannot be absolutely satisfied.

Non-restrictive. It has been shown that the user can customize the system to any level of detail, changing or overriding any of the system defaults. There are no arbitrary restrictions placed on the user. While this allows the user to make selections that exhibit undesirable properties, it is condescending to dictate to the user what is best. The system attempts to suggest functional and potentially harmonious colour schemes, but all decisions of the user take precedence. In particular, users can specify colours using exact values which will not be changed, just as they do with current window systems.

Easily customizable. Sections 5.4.3 and 5.4.5 demonstrate how the system can be easily customized by the user.

Predictability and Performance. The aspects of predictability and performance of the system discussed in Section 5.1 are both functions of the implementation of the window system. In particular, they depend almost entirely on the implementation of the constraint solver. They are addressed in Section 6.2.

Adding colour constraints to a dynamic window system with the goal of assisting with aesthetic colour selection is feasible. By satisfying the fundamental properties of dynamic window systems, and avoiding problems with colour association, a usable system can be implemented. In Chapter 6, such an implementation is presented.

Chapter 6

Implementation

6.1 The NeWS Window System

It was decided at an early stage of this thesis not to implement a new window system. Rather, the capability for using dynamic colour to assist users in making aesthetic colour choices should be added to an existing window system. Furthermore, it was also decided that the window system should be one that runs on one of the many Unix workstations available in the Computer Graphics Laboratory. An obvious choice for the window system might have been X11, due to its widespread popularity on these machines. However, a few limitations of X11, which are discussed below, ruled it out as a candidate. Instead, the NeWS window system was chosen. Schlueter (1990) provides a good overview of the important features of NeWS which make it ideal for doing window system research. Only the particularly relevant points are discussed here. The interested reader is directed to the NeWS 2.1 Manual (Sun, 1990) and the The NeWS Toolkit Reference Manual (Sun, 1991) for more a more in-depth discussion of the NeWS window system.

6.1.1 Why NeWS is Appropriate for Research

There are many feature of NeWS that make it ideal for doing window system research. Most importantly, NeWS is easily extensible because it is controlled by programs written in an Object Oriented extension to the PostScript language[1]. Typical features of a window system, such as windows and menus, are implemented as classes in this language. Therefore, adding additional functionality to the window system can done with a minimum amount of coding by building on existing classes. Furthermore, these programs are interpreted, not compiled. This results in a much faster turnaround in the modify-compile-test cycle when prototyping a new system since the compilation step is eliminated. As well, changes can be made to the window system while it is running, allowing new ideas to be tested easily.

The processes that implement the X11 and NeWS window systems are called window *servers* because they provide a windowing service to other applications. An important feature of the NeWS server is that it provides concurrent execution of multiple lightweight PostScript processes. A process is created for each client application that opens a connection with the window server. Additional processes can be created and destroyed in a straightforward fashion by the client. Typically, the client process(es) handle the user interface of the client

[1] It is assumed that the reader is familiar with object oriented programming terminology. Readers unfamiliar with object oriented programming terminology should refer to Appendix B for a brief overview.

application, this being referred to as the *server-side* of the application. Since PostScript in interpreted, and therefore inherently inefficient, the bulk of an application is written in a more efficient, compiled language and run externally. This is referred to as the *client-side* of the application because the compiled program is a client of the window server. PostScript uses dynamic binding for all references, so references to data objects or methods are resolved at runtime. This means that all of the NeWS operators and methods, including the window system methods, can be changed without requiring recompilation of existing applications.

Tied in with the multiprocess nature of the NeWS window server is the event distribution system. To distill input events to the appropriate applications and to allow NeWS processes to communicate with each other, NeWS provides a powerful event distribution mechanism. A process can express an interest in many types of events, such as those created by keyboard input, mouse movement, window damage, etc. Furthermore, a process can create and distribute any type of event, including new types of events they define themselves. The distribution mechanism is very flexible, allowing event distribution to be controlled in a number of useful ways. For example, by specifying which process is to receive an event, interprocess communication is possible. As well, all events are time-stamped, allowing events to be created for future distribution. These features of the event distribution mechanism are used heavily throughout this implementation.

6.1.2 The Choice between NeWS and X11

For this thesis, however, not only is NeWS the most appropriate window system for the implementation, it is in fact the only known window system with the necessary features. It has already been pointed out that the choice is primarily between NeWS and X11.

The NeWS Toolkit 2.0 (TNT2.0) provides the standard set of PostScript classes that define the NeWS window system. This toolkit provides an appropriate colour interface for the purpose of this thesis. Two logical colours, **Foreground** and **Background**, are defined along with a set of interfaces for modifying and retrieving their values. These two colours already contain some of the semantic information required by this implementation: they are defined to be the background and foreground colours for the application area of a window. More importantly, the allocation and management of the colour lookup table (LUT) entries corresponding to these colours is done by the server, not the application. A client can set the foreground colour, for example, but is unaware of the implementation of this action. Indeed, all the application knows is that the specified foreground colour is used from then on.

Under X11, an application asks the server for specific colour values and is returned an index of an entry in the LUT which contains that colour value. Little semantic information is provided to the window server by the application. Adding additional semantic information would necessitate modifying many applications that are to be used with the dynamic window system.

6.2 The Constraint Solver

The task of the constraint solver is to find an optimal set of colours for the window system within a set of constraints. The behaviour of the dynamic window system is determined by the implementation of its constraint solver. In Section 5.5, it was suggested that the predictability and performance of the window system also depends on the implementation of the constraint solver. The implementation of the constraint solver is based on two conceptual models which together satisfy these concerns: the *distributed-jostling model* and the concept of a *dynamical colour system*.

6.2.1 The Distributed Jostling Model

A concept which greatly improves the efficiency of this implementation is the *distributed-jostling model* of constraint solving. This model was used in Schlueter (1990) to implement the system for perceptual synchronization. A *distributed* constraint solver is possible because finding a globally optimal colour set is equivalent to finding a satisfactory set of colours for each window. Thus, each window in the system participates in the search for an optimal colour set by finding a solution for its colours. More interesting is the *jostling* nature of the constraint solver. When attempting to find a satisfactory set of colours, a window only considers another window if there are constraints between these two windows' colours which are not satisfied. When a window's colours change enough that they are no longer affected by another window, the other window is immediately forgotten. Since not all other windows are considered, a window may come into conflict with previously non-conflicting windows when attempting to find a solution for its colours. A conflicting window notices the conflict and adjusts its colours. This may in turn cause conflicts with other windows, including the window which originally created the problem, requiring that they adjust their colours. The window colours can be viewed as "jostling" for position with their neighbours.

This model is useful for the following reasons. First, the issue of predictability is satisfied because the window colours start changing immediately, providing immediate feedback. Second, performance is improved because iterations of the jostling process are staggered over time, each taking a relatively small amount of the available computing resources. While staggering the iterations of the system increases the time taken to find a solution, this slowness turns out to be a positive feature. Recall from Section 5.3 that a potential problem with dynamically changing colour is that it may destroy the ability of the user to build strong colour associations. One of the ways of countering this problem is to change colours slowly so that the user notices and adapts to them. This is very important, as rapid changes are unnerving.

There are two other notable features of this model for constraint solving in a dynamic window system. First, since there is no global attempt at discovering an optimal colour set, this approach scales better to larger numbers of windows than a centralized algorithm. Second, algorithms that implement this model are simple to implement.

6.2.2 The Dynamical Colour System

Throughout this thesis, the window system is called a *dynamic* window system. One reason is that this window system can, in fact, be modelled by a mathematical dynamical system. It seems reasonable, then, that the constraint solver should be implemented as a simple dynamical system. Instead of viewing a colour in the system as an abstract quantity, consider it to be a physical object. Each colour object exerts repulsive and attractive forces on other colours, with the forces created by the constraints between those colours. Each colour has a position which is its location in colour space. Colour acceleration and colour velocity are derived in the usual manner, using Newton's Second Law.

$$F = ma \tag{6.1}$$

which gives

$$\begin{aligned} \bar{d_0} &= \text{initial position} \\ \bar{v_0} &= \text{initial velocity} \\ \bar{a} &= \bar{F} - \bar{v_i}\mathcal{F}/m \\ \bar{v}_{i+1} &= \bar{v_i} + \bar{a}\Delta t \\ \bar{d}_{i+1} &= \bar{d_i} + \bar{v_i}\Delta t \end{aligned} \tag{6.2}$$

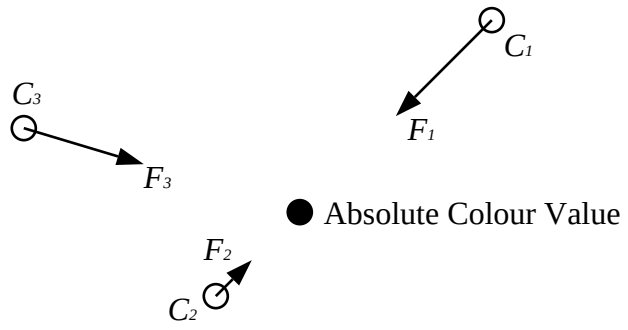


Figure 6.1: The force attracting colours to an absolute colour value act along the line between the colour and the absolute colour value, directed from the colour to the absolute colour value.

where $\overline{d}$ is the colour position, $\overline{v}$ is the velocity of the colour, $\overline{F}$ is the force acting on the colour, m is the mass of the colour, $\mathcal{F}$ is a damping factor, $\overline{a}$ is the acceleration of the colour[2], and Δt is the time interval between time i and time $i+1$. What is meant by colour mass is explained below. Viewing the colours in this manner provides us with an elegant constraint solver based on Newton's Laws, with one caveat which shall be discussed below. The constraints between colours are expressed as forces in a straightforward manner.

There are two kinds of constraints, absolute and relative. An absolute constraint acts as a force between a colour and a specific colour value, directed along the line from the colour's current position to the position of the specific colour value. Forces are either attractive or repulsive. For attractive forces, the farther the colour is from that value, the higher the force. For repulsive forces, the closer the colour is the the value, the higher the force. A situation where a colour is too far from an attractive absolute colour value is illustrated in Figure 6.1. In a sense, it is as if the colour is either anchored to a location in colour space, such as by a spring, or is being repelled from a location in colour space, such as by a magnet of the same polarity.

Relative constraints are expressed similarly, except that the force acts between two colours and is exerted on both of them in equal and opposite directions. There is a relationship between the two colours that is expressed as the constraint. When this constraint is violated, both colours attempt to change their values to satisfy it. As with the previous constraints, the farther the colours are from satisfying it, the higher the force. A situation in which two colours are too far apart is illustrated in Figure 6.2. Even when relative constraints are hard to visualize as forces, they are easy to represent mathematically.

In Chapter 5, constraints were considered to have more or less importance. This concept is easy to accomplish by varying the magnitude of the force vector produced by the constraint. Furthermore, this type of constraint solver automatically relaxes its constraints so that solutions which are closest to satisfying the constraints may be found. Consider the situation when a colour is bordered on all sides by colours which are exerting repulsive forces. While the colour may not be able to move to a position where there are no more forces acting on it, it does move to a position where the *net force* acting on it is zero. This represents a optimal solution which comes closest to satisfying the constraints and occurs naturally with this model.

Another requirement of a dynamic colour system is that some colours be less likely than others to change their values. The concept of colour mass makes this possible. Equation 6.2 shows that increasing the mass of an object

[2]The $\frac{1}{2}a\Delta t^2$ term is omitted because of its relatively small effect.

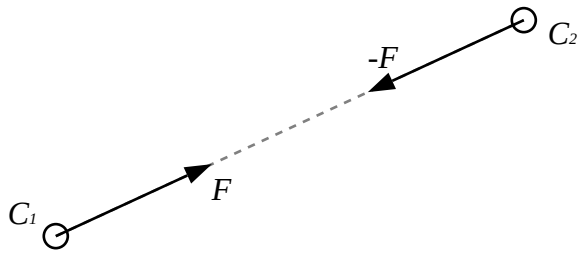


Figure 6.2: The force attracting the two related colours acts along the line between them, directed from each colour toward the other.

decreases its acceleration. When the mass of a colour object is varied, it becomes more or less likely to move when it interacts with other colours. For example, if an exact colour value has been specified for a colour it should never move. This is easily accomplished by giving the colour an infinitely large mass, so all forces result in zero acceleration. Consider another example. If the user has expressed a preference for a certain colour value it should be less likely to move than colours which were chosen randomly by the system. This is accomplished by giving the colour a larger mass that the average colour.

As it turns out, this model fits quite well with the distributed jostling model. Each window is given its own dynamic system in which it puts its colours and its intra-window constraints. When a constraint between one of this window's colours and a colour in another window is violated, that window's colours are added to the local dynamic system along with the constraints in question. During each iteration of the dynamic system, the sum of the forces acting on each local colour is used to determine the acceleration acting on the colour. This acceleration is used to update the velocity and position of the colour using Equation 6.2. If any local colours move, all windows are notified of the change in position. Any external constraints that exert no force on the local colours are discarded, and any external windows whose colours are no longer in conflict with the local colours are discarded.

The final feature of this model is the use of a force similar to friction in damping oscillations. By applying a force against the motion of colours, the system naturally dampens any potential oscillations that could occur. $\mathcal{F}$ is the damping factor in Equation 6.2.

It is mentioned at the beginning of this section that there is one caveat to the statement that this system is based on Newton's Laws. That caveat is that when there are no longer any forces acting on a colour object aside from the force of friction, it immediately stops moving. While this introduces slight discontinuities into the system, this highly damped behaviour is required to ensure that colours change as little as possible to prevent unnecessary damage to user colour associations.

6.3 The NeWS Colour Window Classes

The actual implementation consists of a hierarchy of classes built on top of the TNT2.0 classes. A separate class exists for each major issue that had to be handled during the implementation. This separation of functionality facilitates fast prototyping, provides a logical organization for explaining the system and allow the system to be

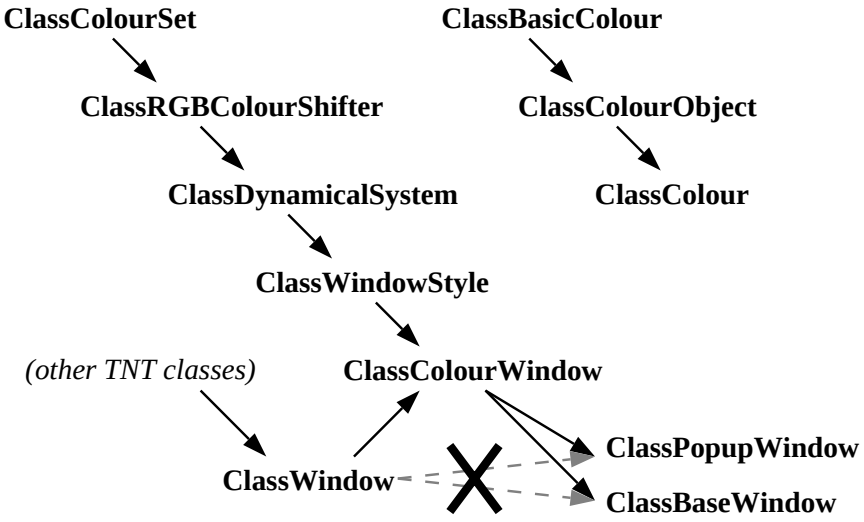


Figure 6.3: The Colour Window System classes are added to the TNT2.0 system by inheriting from the TNT2.0 **ClassWindow** class and modifying the **ClassWindow** subclasses **ClassBaseWindow** and **ClassPopupWindow** to inherit from **ClassColourWindow**.

easily modified for future experimentation. Figure 6.3 shows the class hierarchy of this system, along with an excerpt from the TNT2.0 class hierarchy showing where these classes fit in[3].

A goal of the implementation is to allow the concept of dynamic colour to be added to the NeWS window system in such a way that many existing applications work with the system without needing to be modified or recompiled. A simplification is made to the conceptual model discussed in Chapter 5 in order to achieve this goal. Specifically, only four conceptual colours are supported for each window, as described below.

TNT2.0 provides a standard colour interface which allows applications to specify foreground and background colours. Fortunately, most of the available NeWS applications only use the single foreground and background colours supported via the standard interface discussed earlier. When additional colours are used, applications set them in a variety of ways, usually by using the NeWS **setcolor** operator to directly set the drawing colour just prior to using it. Unfortunately, there is no way to override the **setcolor** operator in an intelligent manner because there is no way to infer the semantic meaning of colours specified in this fashion. Therefore, since changing the applications is to be avoided, the only application colours with which the window system can do anything intelligent are the two colours set through the standard interface. As mentioned above, few of the existing TNT2.0 applications use more than the standard two colours, so most of them work with the system without modification. These two colours are referred to as the *Client* colours throughout the implementation.

Standard TNT2.0 windows share a single set of border colours, which are a medium intensity gray background with a black foreground by default. This restriction is removed in the colour window classes, allowing each window to have its own distinct border colours. These are referred to as the *Border* colours throughout the implementation. These two border colours, and the two client colours mentioned above, comprise the set of four colours for which some semantic information can be implied by the window system.

In the following sections, each of the major classes in the Colour Window System are described.

[3]Throughout NeWS colour is spelled “color”, so references to NeWS routines, classes, types, etc. will be spelled in that way.

6.3.1 ClassBasicColour

NeWS provides an object type called **colortype** to represent colour values. A **colortype** can be created by specifying its coordinates in either RGB or HSB colour space. All of the NeWS colour operators, and most of the class methods, take parameters of type **colortype**. The functionality provided by the **colortype** is not sufficient for this implementation because colours will need to be specified in colour spaces aside from these two.

RGB specification is required. What NeWS calls HSB appears to correspond to the HSV colour space discussed in Chapter 2, but is not adequately described in the NeWS documentation. In any case, the HLS model provides a more intuitive approach to colour specification than HSV, so HLS is used in this implementation. To support both the measuring of contrast and colour difference, colours need to specified by their XYZ and CIELUV coordinates. **ClassBasicColour** provides a replacement for **colortype** that supports all of these colour spaces.

Since the **ClassBasicColour** overrides the NeWS **colortype**, all of the NeWS colour operators and some of the class methods that deal with colour are rewritten to support parameters of this class as well as of **colortype**.

6.3.2 ClassColourObject

This object adds to **ClassBasicColour** the physical characteristics required to use colours in the dynamical system described in Section 6.2.2. The value of the colour is equivalent to its position. Methods are provided to change the mass of the colour and to stop the motion of the object. Force is applied to the object through two methods. **addforce** adds a force vector to the total force acting on the object, as follows:

```
Input: F ( an [h, l, s] force vector )

    MyForce.h += F.h
    MyForce.l += F.l
    MyForce.s += F.s

No Output
```

applyforce first calculates friction as a function of the object's velocity, then applies the total force vector to the object, automatically updating the acceleration, velocity and position of the colour using Equation 6.2, as follows:

```
No Input

    % these are vector operations
    MyAccel := MyForce / MyMass
    MyForce := 0

    if ( magnitude( MyAccel ) < threshold ) then
        stopmotion
        return true
    else
        MyAccel += MyVelocity * Friction / MyMass
        MyPosition += MyVelocity * DeltaTime
        MyVelocity += MyAccel * DeltaTime

        set_my_colour( MyPosition )
        return false

Output: boolean
```

MyAccel, **MyForce**, **MyMass**, **MyVelocity** and **MyPosition** correspond to the acceleration, force, mass, velocity and position of the colour object. If the total external force acting on the colour is close to zero when **applyforce**

is invoked, the colour will immediately stop moving. The **threshold** value determines how small the acceleration must be to be considered zero. While this does not correspond to natural physical laws, it is nevertheless desirable to make sure that colours move no more than is necessary to satisfy the constraints, giving the system a highly damped feeling. **applyforce** returns a boolean value indicating whether the net force acting on the colour was approximately zero, and thus the stability of the colour.

The force is specified as a vector in the HLS colour space. The HLS space was chosen because most of the constraints are expressed in HLS coordinates, as discussed in Section 6.3.6.

6.3.3 ClassColour

The last of the colour objects, and the one used by other classes, is **ClassColour**. It solves the final problem associated with the colour classes, that of overriding the NeWS colour model.

NeWS uses a static colour model. The colour lookup table is filled with a cube of colour that samples the colour gamut at regular intervals. When a colour is requested by an application, the nearest available colour in the colour cube is used. Unfortunately, as was discussed in Section 5.4.1.2 there is no concise definition of colour similarity. As a result, the colour value chosen by NeWS is not necessarily the most appropriate one available. This is especially true when colours are chosen subject to some constraints, such as when they are supposed to be different shades of the same hue.

There are two drawbacks to this scheme. First, and most importantly, when a colour value is requested by this system, it is important that the exact colour value is returned. Otherwise, both the aesthetic and functional colour relationships that are created by this system can be destroyed. Second, in order to change window colours freely, each different colour used by the system must have its own LUT entry. With a static colour model, an application does not have the same LUT entries for its colours after they are changed, requiring it to redraw its windows every time the colours change. To avoid this, the values stored in the LUT entries used by the application are changed. Unfortunately, with the NeWS static colour model, if two colours start with the same value, they will have the same LUT entry. When one of these windows changes its colour value, the other window's value is also changed, which is unsatisfactory. Fortunately, current versions of NeWS allow dynamic colour maps to be created. By forcing the window system to use a dynamic colour map instead of the static one it uses by default, LUT entries are allocated for the exclusive use of particular colours, solving this problem.

A serious limitation is the small size of colour lookup tables. Current systems have 256 entries in their LUT's, which is sufficient for static colours, but with dynamic colours all the entries tend to be used quite quickly. Each window in the window system has two foreground/background colour pairs. However, since TNT draws all of its interface components using an embossed, three dimensional appearance, each of these conceptual colour pairs actually requires five LUT entries. Thus, a typical window in this system requires ten LUT entries. If the three dimensional effects are turned off, each window still requires four colours. To handle this problem, the **ClassColour** object makes use of the idea of *active* and *inactive* colours. When a colour needs to be used, the window system attempts to activate it, causing the **ClassColour** object to attempt to allocate a LUT entry for its exclusive use. If this fails, the **ClassColour** object uses a default LUT entry which it does not allow to be modified. Therefore, when the system runs out of LUT entries, applications run but use a default set of colours instead of their own. Fortunately, as hardware becomes more powerful, this problem will disappear. For example, systems already exist with 4096 LUT entries, such as the Silicon Graphics IRIS workstations.

6.3.4 ClassColourSet

The dynamic colours used by a window are managed by **ClassColourSet**. Each colour is given a symbolic name, indicative of its semantic meaning, which is used to store a reference to the colour in a dictionary called **ColourSet**.

The four major colours automatically created by the system are labeled **BBG** (Border Background), **BFG** (Border Foreground), **CBG** (Client Background) and **CFG** (Client Foreground).

The concept of active and inactive colours introduced by the **ClassColour** object raises two issues which are dealt with by this class. First, if a window has more than one colour, either all or none of them are active. Allowing a subset of the window colours to be active would increase the complexity of the rest of the system without providing any clear benefit to the user. Consider, for example, that most of the window's colours are used to create three dimensional effects for the various interface objects. If any of the colours in that set are inactive, and thus uses its default colour value, all of them must assume the default colour values or the three dimensional appearance of the window is disrupted. Grouping the colours allows them to be activated and deactivated as a set. Also, the semantic meaning of each of these colours is known by **ClassColourSet**, so reasonable default values are provided when colours can not be allocated; black for the foreground colours and shades of gray for the background and three dimensional effects. Thus, windows that can not allocate their colours have an achromatic appearance that distinguishes them from the rest of the windows in the system.

The second problem is that if all of the LUT entries are allocated and a window deactivates its colours, any windows using the default colours should be notified so they can reattempt to activate their colours. To facilitate this notification, **ClassColourSet** defines a **ColourSegFreed** event which is distributed when the window's colours are deactivated. Similarly, when a window is unable to activate all of its colours, it expresses an interest in the **ColourSegFreed** event. To ensure that starvation does not occur when multiple windows are competing for the same small set of LUT entries, *exclusive events* are used. These events are only distributed to one interested process at a time and are used for synchronization because only one window at a time will receive the event and attempt to activate their colours. This prevents problems such as *The Dining Philosophers Problem* (Tanenbaum, 1986) from occuring and guarantees that one of the windows will activate its colours if there are enough LUT entries.

6.3.5 ClassColourShifter

A potential problem with changing colours via a distributed jostling model is that colours are not guaranteed to take a straight or smooth path to their final colour value. Analogous behaviour was seen in Schlueter's system when the windows appeared to jiggle around before settling on their final positions. The movements of the windows in Schlueters system were, however, relatively small. Even though the often vigorous jiggling was distracting, the user could see why they moved as they did. Colour movements are not necessarily as small and colour relationships are not always as obvious. If the colours are allowed to jiggle as vigorously, they inevitably appear to change abruptly in random ways causing jarring visual effects.

It is desirable, therefore, to have the colours shift gradually from their current values to the new ones. **ClassColourShifter** provides this ability in a simple fashion. When the system decides to changes a colour value, the destination value is set inside of this class and the colour slowly changes to that value. As the jostling causes the colour value to change, the destination of the colour is changed and its route altered. Therefore, many quick changes in the destination value of the colour will not be as distracting, as the colour's actual value will move very little between each change in its destination.

6.3.6 ClassColourConstraint

ClassColourConstraint defines the methods common to all constraints. Using this class, it is possible to create all of the constraints needed for this thesis. It is not intended to be instantiated, but rather to provide a template for the creation of specific types of colour constraints.

As described in Section 6.2.2, there are two forms of constraints, absolute and relative. All of the constraints operate on two operands, which are instances of either the **ClassColour** class or the NeWS **colortype**. Recall that **ClassColour** is a subclass of **ClassColourObject**, giving instances of **ClassColour** an understanding of the physical concepts of mass, force, etc. that are necessary for the constraints to exert force on them. A **colortype**, on the other hand, is a elementary NeWS type with no knowledge of the necessary physical concepts. They are useful, however, for specifying exact colour values that will not change. Since all forces result in zero acceleration when applied to an object of infinite mass, by viewing **colortype** operands as colour objects with infinite mass the constraints do not need to apply force to them.

The constraints are modelled as forces between two colours. If the constraint is satisfied, there is no force exerted on the colours. If the constraint is not satisfied, there is a force exerted on the colours, pushing them toward a satisfactory state. How this force is determined is a function of the individual constraints, but the magnitude of the force is always proportional to the distance the colours are from positions that satisfy the constraint. Furthermore, the force is exerted bidirectionally, affecting both colours in equal and opposite directions. However, since the constraint solver is distributed, with each window changing only its local colours, only intra-window constraints actually have their force applied to both colours. If two windows have conflicting colour values, each window applies the same constraint to the two conflicting colours. If the resulting force is applied to both colours in both windows, each colour will have the force applied twice. However, following the principle that a window will only change its local colours, the forces resulting from inter-window constraints are only applied to the local colours, avoiding this problem. A method is provided to inform a constraint if it should be **bidirectional?** or not.

As discussed in Section 6.2.2, some constraints are more important than others. Therefore, a **setimportance** method is provided to set the **Importance** of the force. The value specified is used by the constraint to determine the magnitude of the force. For all of the forces described below, the magnitude of the force vector is simply multiplied by **Importance** value, which defaults to 1.

Many of the constraints in the system, especially the functional constraints, are repeated occurances of a small number of constraints applied to different windows. For example, every pair of foreground and background colours in the window system has a contrast constraint acting on it, yet there are only two distinct constraints. One is between the border foreground and background colours, the other is between the client foreground and background colours. To take advantage of situations such as this, the colours that a constraint acts on are denoted either using the symbolic names defined in **ClassColourSet** or by a NeWS **colortype**. When any of the methods of a constraint are invoked, the **ColourSet**s containing the appropriate window's **ClassColour** objects are provided and the constraint retrieves the needed colours. For example, an instance of a constraint may affect the **BBG** and **BFG** colours. When one of this constraint's methods are invoked, the **BBG** colour is looked up in the first **ColourSet** and the **BFG** colour is looked up in the second. The **setoperand1** and **setoperand2** methods are provided to set the name or colortype of each operand.

There are three methods provided by all constraints, which correspond to the three tasks a constraint may need to perform: **check**, **apply** and **pickrandom**. The **check** method does not apply any force to the affected colour, but returns a boolean result which indicates whether the constraint is satisfied. The **apply** method first checks to see if the constraint is satisfied. If it is not satisfied, force is applied to the colours to push them toward a satisfactory state. The **pickrandom** method is used when new colours must be generated for a window, as described in Section 6.3.8. It assumes the second operand's colour is valid and picks a random colour value for the first operand such that the constraint is satisfied.

The general algorithm used by all constraints for the **check** and **apply** routines is as follows:

```
Input: op1, op2 ( type ClassColourSet or colortype )

    if type(op1) = ClassColourSet then
        op1 := lookup key operand1 in dictionary op1
```

```
        if type(op2) = ClassColourSet then
            op2 := lookup key operand2 in dictionary op2

        if type(op1) = ClassColour or
           ( type(op2) = ClassColour and bidirectional ) then
            { WORK }
        else
            return true

    Output: boolean
```

where { **WORK** } in the **check** routine is:[4]

```
        return (( F := getforce(op1, op2) ) = 0 )
```

and in the **apply** routine is:

```
        F := getforce(op1, op2)
        if type(op1) = ClassColour then
            addforce(op1, F)

        if type(op2) = ClassColour and bidirectional then
            addforce(op2, negative(F))

        return ( F = 0 )
```

getforce is calculated differently for each type of constraint, each of which will be explained in the corresponding section. **addforce** is the **addforce** routine discussed in Section 6.3.2.

The **pickrandom** routine is:

```
    Inputs: op1, op2 ( type ClassColourSet or colortype )

        if type(op1) = ClassColourSet then
            op1 := lookup key operand1 in dictionary op1

        if type(op2) = ClassColourSet then
            op2 := lookup key operand2 in dictionary op2

        if type(op1) = ClassColour then
            colour := pickrand(op1,op2)
            setcolour(op1, colour)

    Output: none
```

As with **getforce**, **pickrand** is calculated differently for each type of constraint and explained in the appropriate section.

Three types of constraints have been implemented which encompass all of the constraints discussed in the thesis. These constraints are **ClassCCVariation**, **ClassCCContrast** and **ClassCCDistance**. Two other constraints, **ClassCCAnalogousVariation** and **ClassCCWindowDistance**, were implemented to demonstrate how more interesting constraints can be built by subclassing from these three basic constraints. Each of these constraints is discussed below.

[4] The **check** routines for the existing constraints have been coded more efficiently by only doing as much work as is necessary to determine if the force would be non-zero. The details are omitted for simplicity.

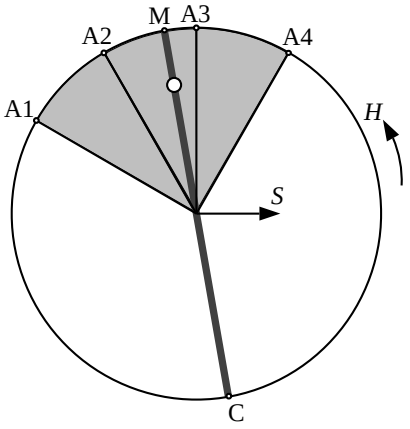


Figure 6.4: The relationships between the hues for the simple colour schemes described in Section 2.4. Hue and saturation are shown, with the lightness axis perpendicular to the page. The colour on which the scheme is based is represented by the open circle. Monochromatic variations have hue **M**. Complementary colours have hue **C**. Analogous colours either have **A1** $\geq$ hue $\geq$ **A3** or **A2** $\geq$ hue $\geq$ **A4**.

6.3.6.1 ClassCCVariation

All the aesthetic constraints discussed in this thesis are implemented using the **ClassCCVariation** class. Each constraint defines one colour as a *variation* of an other one. For example, a monochromatic constraint defines one colour to have the same hue as an other, making the former colour a monochromatic *variation* of the latter.

For these constraints, the HLS colour coordinate system is used. The three HLS coordinates — hue, lightness and saturation — are defined in the range $[0, 1]$. The hue value represents a continuous circle of hues, with hue values of 0 and 1 being equivalent. The lightness range is also treated specially. Noting Gerritsen's work (Section 2.2.6), it is often desirable to specify the perceived brightness of a colour rather than the abstract lightness value. For example, a user may desire that all of their window backgrounds have the same perceived brightness. To support this, the lightness range specification can apply to either the HLS lightness coordinate or the XYZ Y coordinate of the colour.

Variations are defined by providing either an *relative* or *absolute* range of acceptable values for each of the coordinates. A relative range indicates the amount the two colours can differ in the coordinate, specifying the value of the first colour operand relative to the value of the second. For example, if a relative range of $[-.25, +.15]$ is specified for the saturation and the second colour operand has a saturation of .3, the first colour's saturation could fall anywhere from .05 to .45 units. An absolute range, on the other hand, specifies the possible values of the first colour operand independently of the second. For example, if the lightness was specified by the absolute range $[.5, .6]$, the first colour would have a lightness of between .5 and .6 irregardless of the lightness of the second colour.

Using this class, the following interesting constraints can be created:

- **Monochromatic colours.** Recall from Section 2.4 that a colour is a monochromatic variation of another colour if they both have the same hue, as shown in Figure 6.4. To create a monochromatic constraint, a **ClassCCVariation** constraint is applied to the colours with a relative hue range of $[0, 0]$ and absolute lightness and saturation ranges of $[0, 1]$. This condition constrains the hues to be the same but does not affect

the saturation or lightness. By further refining the brightness and saturation ranges, more specific results can be obtained. For example, consider the default colours scheme suggested in Section 5.4.4. There were two background colours in that scheme:

- *border background.* Any hue, a saturation of 25%, a lightness of 50%.
- *client background.* Same hue and saturation as the border, a lightness of 90%.

They could be created with the following constraints:

- *border colour.* Absolute hue range $[0, 1]$, absolute saturation range $[.25, .25]$, absolute lightness range $[.5, .5]$.
- *window interior.* Relative hue and saturation range $[0, 0]$, relative lightness range $[+.4, +.4]$.

- **Complementary colours.** Recall from Section 2.4 that two colours are complementary if their hues lie exactly opposite each other on the circle of hues. As mentioned above, the hue value represents a circle. Effectively, this means addition and subtraction of hue values is performed modulo 1 so that hue values wrap around the ends of the range properly. Therefore, specifying a constraint between two colours with a relative hue range of $[.5, .5]$ constrains them to be complementary colours. The saturation and brightness are handled in the same fashion as for monochromatic colours.

The force exerted by the **ClassCCVariation** constraint is determined independently for each of the H, L and S coordinates, the magnitude and direction being proportional to the distance of the coordinate from the closest edge of the range of satisfactory colour values. This is important. The first colour operand of this constraint is only partially defined in relation to the second colour operand. Specifically, any of the coordinates that are constrained to absolute ranges are independent of the second colour operand. The force exerted on the first operand is determined by adding the three independent forces. However, the force on the second operand is determined by adding only those forces whose associated coordinate is constrained by a relative range, since these are the only coordinates which interact with the colour value of the first operand.

Each component of the force can have a different importance specified for it, allowing different aspects of the relationships expressed via these constraints to have different conceptual importance. For example, a relative hue range of $[0, 0]$ might be specified because the user wishes the colours to be monochromatic variations of each other. By giving the hue relationship a high importance, the likelihood that the constraint will be satisfied is increased. However, it may not be desirable to assign the lightness or saturation components of the constraint an equally high importance. For example, a relative lightness range of $[+.25, +.25]$ might be specified so that one colour is lighter than the other. The importance of this aspect of the constraint may be significantly less than the importance of the hue aspect.

In Section 6.3.6, the **check** and **apply** routines were defined in terms of a **getforce** function. For **ClassCC-Variation**, that function is:

```
Inputs: op1, op2 ( type colortype )

    % calculate the saturation force component
    sat := sat(op1)
    if SaturationMode = Relative then
        sat := sat - sat(op2)

    if sat >= SatRangeMin then
        if sat <= SatRangeMax then
            F.s := 0
        else
            F.s := SatRangeMax - sat
```

```
    else
        F.s := SatRangeMin - sat

    % calculate the lightness force component
    light := light(op1)
    if LightnessMode = Relative then
        light := light - light(op2)

    if light >= LightRangeMin then
        if light <= LightRangeMax then
            F.l := 0
         else
            F.l := LightRangeMax - light
    else
        F.l := LightRangeMin - light

    % calculate the hue force component
    hue := hue(op1)
    if HueMode = Relative then
        hue := (hue - hue(op2)) mod 1

    if hue >= HueRangeMin or hue <= HueRangeMax then
        F.h := 0
    else if HueRangeMin < HueRangeMax then
        % median is half the size of the
        % area outside the hue range
        median := (1 - HueRangeMax + HueRangeMin)/2
        if hue < HueRangeMin then
            % if hue is within median of min,
            % then hue is closest to min
            if median > (HueRangeMin - hue) then
                F.h := HueRangeMin - hue
            else
                F.h := hue + 1 - HueRangeMax
        else
            % if hue is within median of min,
            % then hue is closest to min
            if median > (hue - HueRangeMax) then
                F.h := HueRangeMax - hue
            else
                F.h := HueRangeMin + 1 - value
    else
        % if HueRangeMin > HueRangeMax, the value hue
        % range wraps around the end.
        median := (HueRangeMin - HueRangeMax)/2 +
                    HueRangeMax
        if median < hue then
            F.h := HueRangeMin - hue
        else
            F.h := HueRangeMax - hue

    F.s *= SatImportance
    F.l *= LightImportance
    F.h *= HueImportance

Output:  F ( the force vector )
```

In addition, **ClassCCVariation** redefines the **negative** routine as follows:

```
Input: F ( a force vector )

    if SaturationMode = Relative then
        F.s *= -1
    else
        F.s := 0

    if LightnessMode = Relative then
        F.l *= -1
    else
        F.l := 0

    if HueMode = Relative then
        F.h *= -1
    else
        F.h := 0

Output: -F ( the force vector )
```

As explained above, only those coordinates of the second operand that influence the first operand are to have force applied to them.

The **pickrandom** routine was defined in terms of a **pickrand** function. For **ClassCCVariation**, that function is:

```
Inputs: op1, op2 ( of type colortype )

    colour.s := SatRangeLength * random + SatMin
    if SaturationMode = Relative then
        colour.s += sat(op2)
    colour.s := colour.s mod 1

    colour.l := LightRangeLength * random + LightMin
    if LightnessMode = Relative then
        colour.l += light(op2)
    colour.l := colour.l mod 1

    colour.h := HueRangeLength * random + HueMin
    if HueMode = Relative then
        colour.h += hue(op2)
    colour.h := colour.h mod 1

Output: colour ( of type colortype )
```

6.3.6.2 ClassCCAnalogousVariation

ClassCCAnalogousVariation is a subclass of the **ClassCCVariation** class, demonstrating how more interesting aesthetic constraints can be created. This class overrides the hue specification of **ClassCCVariation** so that the hues of the two colours vary within an analogous colour range. It is necessary to create a more specialized class because analogous hue cannot be defined by the simple linear relationships possible with **ClassCCVariation**.

Recall from Section 2.4 that an analogous colour scheme can be defined by dividing the colour circle into twelve segments and selecting any neighbouring pair of segments. All colours that fall within this pair of segments

define a set of analogous colours. The colour on the border of two segments can be considered to fall in both segments, but is restricted to one of them for simplicity. If the hue parameter defines a clockwise circle, the hues on the borders of segments are included only in the segment on their clockwise side. Therefore, any given colour value falls within two analogous colour schemes, as shown in Figure 6.4.

By default, the constraint permits the hue of the first colour operand to fall anywhere in the range of hues possible in either of the analogous schemes that could be defined by the hue of the second colour operand, allowing the widest range of hues. When a single **ClassCCAnalogousVariation** constraint is being used to relate two colours, this is a reasonable approach. However, when more than two colours are being related with **ClassCCAnalogousVariation** constraints with the intent of having them all be analogous to each other, either a constraint must be placed between every pair of colours, or they all must be related to a common colour by one of the two analogous schemes. Otherwise, the colours are not guaranteed of falling in the same analogous colour scheme. The former approach is desirable when a small number of colours are being related, since it allows the most flexibility in colour selection. However, the number of constraint required is $\binom{N}{2} = N(N-1)/2$ where N is the number of colours. When the number of colours is large, the second approach is more desirable, because only $N-1$ constraints are needed. Additionally, the second approach allows more control over which analogous colour scheme is used. If only one of the schemes is desired, the user may specify which of the two schemes is to be used. The pseudocode for the calculation of possible hues is as follows:

```
Input:  hue2 ( the hue of operand2, range [0,1] )

    % the hues are divided into 12 analogous ranges
    sector := integer( ( hue2 * 12 ) mod 12 )

    % "Either" selects three sectors, the other
    % methods select the left or right pair
    if Analogous_Scheme = Either then
        Range_min := sector - 1
        Range_max := sector + 2
    else if Analogous_Scheme = Left then
        Range_min := sector
        Range_max := sector + 2
    else
        Range_min := sector - 1
        Range_max := sector + 1

    % convert back to [0,1] ranges
    Range_min := ( Range_min mod 12 ) / 12
    Range_max := ( Range_max mod 12 ) / 12

Output: Range_min, Range_max
```

ClassCCAnalogousVariation is implemented by overriding the **apply**, **check** and **pickrandom** interface methods of **ClassCCVariation**. The new methods determine the appropriate hue range for the analogous colour scheme defined by the colour value of the second operand, use it to set the **ClassCCVariation** hue range and finally invoke the superclass method they override. Since each instance of the constraint can be used multiple times, the code between setting the hue range and invoking the superclass method is a critical section (Tanenbaum, 1986). Synchronization is provided by enclosing the setting of the superclass hue range and the invocation of its **check**, **apply** or **pickrandom** method with a monitor.

As mentioned above, the class definition for **ClassCCAnalogousVariation** is quite short, demonstrating how simple new constraints are to create. It is included in Appendix C to illustrate that point.

6.3.6.3 ClassCCDistance

The **ClassCCDistance** constraint is used to specify a distance relationship between two colours. As suggested in Section 5.4.1.2, the distance between two colour values is calculated as the Euclidean distance in the CIELUV colour space. Since both maximum and minimum distance constraints are needed, the constraint can be used to either attract colours so they attempt to stay within a certain distance or repel colours so they attempt to stay beyond a certain distance. Methods are provided to set the **Distance** threshold and to specify whether this constraint should exert an attractive or repulsive force by defining the boolean parameter **Attract?**. The magnitude of the force is proportional to the distance that the value is from the threshold, adjusted by the importance of the constraint. The direction of the force is collinear to the two colour values, either directed towards the colours or away from the colours depending on if the constraint is attractive or repulsive.

In Section 6.3.6, the **check** and **apply** routines were defined in terms of a **getforce** function. Before defining this function, the **luvdistance** function is defined as the Euclidean distance between the two colours as follows:

```
Inputs: colour1, colour2 ( type colortype )

    luv1 := LUV(colour1)
    luv2 := LUV(colour2)

    distance := square_root( ( luv2.l-luv1.l )^2 +
        ( luv2.u-luv2.u )^2 + ( luv2.v-luv2.v )^2 )

Outputs: distance
```

Using this function, the **getforce** routine for **ClassCCDistance** is:

```
Inputs: op1, op2 ( of type colortype )

    Force := Distance - luvdistance( op1, op2 )
    if Attract? then
        Force := -Force

    if Force < 0 then
        F := 0
    else
        % scale the force to a more resonable value
        % which has been chosen experimentally
        Force /= 80
        Force *= Importance

        % now, calculate the vector between the two colours
        vector := HLS(op1) - HLS(op2)
        if Attract? then
            vector := -vector

        % make sure the hue is taking the shortest route
        if vector.h > .5 then
            vector.h -= 1
        if vector.h < -.5 then
            vector.h += 1

        % if the vector length is close to zero, don't
        % use it.  PickRandDir returns a unit vector
        % pointing in a random direction
        if LENGTH(vector) < .001 then
            unit := PickRandDir()
```

```
        else
            unit := vector/LENGTH(vector)

        F := unit * Force

Output:  F ( the force vector )
```

The **pickrandom** routine was defined in terms of a **pickrand** function. For **ClassCCDistance**, that function is defined as:

```
Inputs: op1, op2 ( of type colortype )

    if Attract? then
        colour := op2
    else
        vector := PickRandDir() * MinDistance

        % hue wraps around the end
        colour.h := op2.h + vector.h mod 1

        % lightness and saturation are truncated
        colour.l := op2.l + vector.l
        if colour.l > 1 then
            colour.l := 1
        elseif colour.l < 0 then
            colour.l := 0

        colour.s := op2.s + vector.s
        if colour.s > 1 then
            colour.s := 1
        elseif colour.s < 0 then
            colour.s := 0

Output: colour ( of type colortype )
```

This constraint is used to create many relationships, examples of which are given below.

Preferred Colour Specification. Although it is possible to specify the value of a colour as an exact value that does not change, there are times when a particular value is preferred, but the colour can change if other constraints are acting on it. For example, if a user modifies a colour manually, a constraint of this type is created by specifying an attractive force with a distance threshold of zero.

Similar Colours. When windows are grouped together as described in Section 5.4.1.2, their colours should be similar. Because colour similarity is not well understood, a fairly arbitrary threshold has been chosen within which colours are taken to be similar. From Section 5.4.1.2 it can be seen that distances as small as five OSA units can provide good colour separation. Therefore, the distance threshold should be smaller than this. A reasonable choice is the OSA nearest neighbours, which have a distance of $\sqrt{3}$ and 2. From Appendix A, it can be seen that this corresponds to 15 to 20 CIELUV units. Fifteen was chosen to err on the conservative side. Therefore, this constraint is created by specifying an attractive force with a distance threshold of fifteen.

Slightly Different Colour. Unless window colours are explicitly constrained to be the same, they should be distinguishable, even if they are constrained to be similar as described above. From Appendix A we see that an

CIELUV difference of approximately ten units is a reasonable value to use for slight difference. Therefore, this constraint is created by specifying a repulsive force with a distance threshold of ten. It is also given a fairly high importance so that the colours do not come significantly closer to one another than ten units.

Significantly Different Colour. When windows should be visually different, some of their colours are required to be significantly different. As discussed in Section 5.4.1.2, an CIELUV distance of approximately eighty units is a reasonable value to use. Therefore, this constraint is created by specifying a repulsive force with a distance threshold of eighty.

6.3.6.4 ClassCCWindowDistance

ClassCCWindowDistance, like **ClassCCAnalogousVariation**, is an example of how to create more powerful constraints based on the three basic types. **ClassCCWindowDistance** uses **ClassCCDistance** to create a simple window distance constraint that can be used to repel or attract windows, just as **ClassCCDistance** can repel or attract individual colours.

Window distance is defined to be the difference between either the border background colours or the client background colours of the two windows. The pair with the highest chromatic component is chosen, since these colours represent the predominant window colour. The heuristic used to calculate the chromatic component of a colour value is

$$Cr \quad = \quad 2\ \left(.5 - |L - .5|\right)\ S \tag{6.3}$$

where L and S are the HLS lightness and saturation coordinates of the colour. Since the HLS saturation coordinate represents the percentage of the maximum saturation for a given lightness value, and colour values are less saturated the farther their lightness value is from .5, Equation 6.3 is a measure of the absolute saturation of a colour independent of the lightness.

ClassCCWindowDistance is implemented similarly to **ClassCCAnalogousVariation** by overriding the **apply**, **check** and **pickrandom** interface methods of **ClassCCDistance**. Each of these routines determines which pair of colours should be used, sets the the operands of its **ClassCCDistance** superclass appropriately and invokes the corresponding superclass method. Just as with **ClassCCAnalogousVariation**, each instance of the constraint can be used multiple times, so a monitor is used for synchronization.

Like **ClassCCAnalogousVariation**, this constraint is very simple and is included in Appendix C.

6.3.6.5 ClassCCContrast

The **ClassCCContrast** constraint is used to ensure that a pair of colours, one a background colour and one a foreground colour, maintain a specified contrast. This constraint is implemented using the contrast metric derived in Chapter 4. Since every pair of background and foreground colours in the system has a contrast constraint applied to it, some care is taken with efficiency of the implementation. While there are many variables in the equations that describe the contrast metric, most of them are set once when a specific instance of *ClassCCContrast* is created. For example, the black level and pixel bleed are functions of the monitor and the pixel densities are functions of the application using the two colours. The only parameters that vary between each application of the constraint are the luminances of the two colours. Thus, Equations 2.15, 4.7 and 4.8 are rearranged to isolate these values.

As discussed in Chapter 4, only luminance contrast is considered, since chromatic contrast is not well understood. While luminance contrast may not be necessary for legibility, it is sufficient. As with other constraints,

all parameters are specified assuming the first operand is constrained to the second. Since the calculation of the luminances L'_{bg} and L'_{fg} from Equations 4.5 and 4.5 are different, the boolean **Background?** flag specifies whether the first operand is the foreground or background colour. Furthermore, the desired **Polarity** of the colours may be specified so that either light text on a dark background, or the reverse, can be absolutely required. The first colour can be constrained to be **Lighter** than the second, **Darker** than the second or to be **Either** polarity. If either polarity is acceptable, whichever one is more easily satisfied is used.

When the colours do not have the required luminance contrast, a force is applied which is proportional to the difference between the current contrast and the required contrast, and is parallel to the lightness axis of the colour space.

Before giving the pseudocode algorithms used in the contrast constraint, the following constants are derived from the equations in Chapter 4:

```
BGfg := (1 - BGPixelDensity) * PixelBleed
FGfg := PixelBleed * FGPixelDensity + 1
FGbg := (1 - FGPixelDensity) * PixelBleed
BothBL := (1 + PixelBleed) * BlackLevel
```

where **FGPixelDensity** is the percentage of foreground pixels next to another foreground pixel and **BGPixelDensity** is the percentage of background pixels next to another background pixel. These can be changed depending on the application, but the defaults assume a text based application with thin fonts. **PixelBleed** is the amount of pixel bleed. The default is the optimal value 62%. **BlackLevel** is the black level of the monitor. The default value is 2.5%.

A routine called **getcontrast** is defined which calculates the contrast using the following algorithm:

```
Inputs: Y1, Y2 ( the luminances of the two colours )

    if Background? then
        Ybg := Y1, Yfg := Y2
    else
        Ybg := Y2, Yfg := Y1

    Lfg := Yfg * FGfg + Ybg * FGbg + BothBL
    Lbg := Yfg * BGfg + Ybg * BGbg + BothBL

    Lmax := MAX( Lfg, Lbg )
    Lmin := MIN( Lfg, Lbg )

    C := ( Lmax - Lmin ) / ( Lmax + Lmin)
    Ok? := C > Contrast

Outputs: C, Ok? ( the contrast and if it is sufficient )
```

Using this function, the **getforce** routine for **ClassCCContrast** is:

```
Inputs: op1, op2 ( of type colortype )

    Y1 := XYZ( op1 ).y, Y2 := XYZ( op2 ).y
    F.h := F.s := 0

    (C, Ok?) := getcontrast( Y1, Y2 )
    if Polarity = Either then
        if not Ok? then
            % check if the current polarity can satisfy
```

```
                % the required contrast
                if Y2 < Y1 then
                    % compare Y1 to White
                    (Ccurr, Ok?) := getcontrast( Y1, 1 )
                else
                    % compare Y1 to Black
                    (Ccurr, Ok?) := getcontrast( Y1, 0 )

                if not Ok? then
                    % see if the other polarity will work
                    if Y2 > Y1 then
                        % compare Y1 to White
                        (Cother, Ok?) := getcontrast( Y1, 1 )
                    else
                        % compare Y1 to Black
                       (Cother, Ok?) := getcontrast( Y1, 0 )

                    % aim for the higher contrast.
                    if Ccurr < Cother then
                        if Y2 > Y1 then
                            F.l := 1
                        else
                            F.l := -1
                    else
                        if Y2 > Y1 then
                            F.l := Ccurr - Contrast
                        else
                            F.l := Contrast - Ccurr
                else
                    if Y2 > Y1 then
                        F.l := Ccurr - Contrast
                    else
                        F.l := Contrast - Ccurr
                Ok? := false
            else
                F.l := 0
                OK? := true
        else
            if Polarity = Lighter then
                if Y2 <= Y1 then
                    if not Ok? then
                        F.l := Contrast - C
                    else
                        F.l := 0
                else
                    % it's the wrong polarity, force toward white
                    F.l := 1
                    Ok? := false
            else
                if Y2 >= Y1 then
                    if not Ok? then
                        F.l := C - Contrast
                    else
                        F.l := 0
                    endif
                else
                    % it's the wrong polarity, force toward black
                    F.l := -1
                    Ok? := false

    Output: F ( the force vector )
```

The **pickrandom** routine was defined in terms of a **pickrand** function. Unlike the **getforce** function, the **pickrand** function must calculate a minimum or maximum luminance threshold for the first colour based on the luminance of the second colour. Using this threshold, **pickrand** then selects a luminance at random from the range of satisfactory values.

In order to isolate the luminances of the foreground and background colours from Equations 2.15, 4.7 and 4.8 to determine a luminance threshold, the **Polarity** and **Background?** values must be known. There are four possible equations for calculating the luminance threshold, corresponding to the different values of these two parameters.

Throughout the following equations, the parameters of the contrast metric from Chapter 4 are denoted as follows:

$$
\begin{aligned}
0 &\leq \mathbf{C} \leq 1 \\
0 &\leq PD_f \leq 1 \\
0 &\leq PD_b \leq 1 \\
0 &\leq \sigma \leq 1 \\
0 &\leq BL \leq 1
\end{aligned}
$$

C is the required level of contrast. The default value is 40%. PD_f is the percentage of foreground pixels next to another foreground pixel. Similarly, PD_b is the percentage of background pixels next to another background pixel. σ is the amount of pixel bleed. *BL* is the black level of the monitor.

If **Background?** = true and **Polarity** = **Lighter**, the threshold is the minimum value for the luminance of the background colour:

$$
\begin{aligned}
C_{f_1} &= ((PD_f - PD_b + 1)\,\sigma + 1)\,\mathbf{C} + (PD_f + PD_b - 1)\,\sigma + 1 && (6.4)\\
C_{b1} &= ((PD_f - PD_b - 1)\,\sigma - 1)\,\mathbf{C} + (PD_f + PD_b - 1)\,\sigma + 1 && (6.5)\\
C_B &= (2\sigma + 2)BL && (6.6)\\
C_1 &= C_{f_1}/C_{b1} && (6.7)\\
C_2 &= (C_B\,\mathbf{C})/C_{b1} && (6.8)\\
Y_t &= Y_b \\
&= C_1 Y_f + C_2 && (6.9)
\end{aligned}
$$

If **Background?** = false and **Polarity** = **Darker**, the threshold is the maximum value for the luminance of the foreground colour:

$$
\begin{aligned}
C_3 &= (C_B\,\mathbf{C})/C_{f_1} && (6.10)\\
Y_t &= Y_f \\
&= Y_b/C_1 - C_3 && (6.11)
\end{aligned}
$$

Similarly, if **Background?** = true and **Polarity** = **Darker**, the threshold is the minimum value for the luminance of the background colour:

$$
\begin{aligned}
C_{f_2} &= ((PD_f - PD_b + 1)\,\sigma + 1)\,\mathbf{C} - (PD_f + PD_b - 1)\,\sigma - 1 && (6.12)\\
C_{b2} &= ((PD_f - PD_b - 1)\,\sigma - 1)\,\mathbf{C} - (PD_f + PD_b - 1)\,\sigma - 1 && (6.13)\\
C_4 &= C_{f_2}/C_{b2} && (6.14)\\
C_5 &= (C_B\,\mathbf{C})/C_{b2} && (6.15)\\
Y_t &= Y_b \\
&= C_4 Y_f + C_5 && (6.16)
\end{aligned}
$$

If **Background?** = false and **Polarity** = **Lighter**, the threshold is the maximum value for the luminance of the foreground colour:

$$C_6 = (C_B \mathbf{C})/C_{f_2} \quad (6.17)$$

$$Y_t = Y_f = Y_b/C_4 - C_6 \quad (6.18)$$

All of the subscripted C values are constants that are recalculated whenever the values of **C**, P_f, P_b, σ or *BL* are changed. Thus, the calculation of the threshold value requires only one multiplication and one division when the constraint is being applied to two colours. Equations 6.9 and 6.18 are used to define a function **getlighter** which returns a lighter luminance threshold. Similarly, Equations 6.11 and 6.16 are used to define a function **getdarker** which returns a darker luminance threshold. Using these two functions, the **pickrand** function for **ClassCCContrast** is defined as follows:

```
Inputs: op1, op2 ( of type colortype )

    Y2 := XYZ( op2 ).y
    TL := getlighter( Y2 )
    TD := getdarker( Y2 )

    % if Polarity is Either AND
    %    only one is in range AND it's TL, or
    %    pick a random one and it's TL
    % or Polarity is Lighter
    if ( ( Polarity = Either ) and
         ( (( TL <= 1 ) exclusive or ( TD >= 0 )) or
           random( 0, 1 ) <= .5 ) ) or
       ( Polarity = Lighter ) then
        if TL > 1 then
            colour.l := 1
        else
            colour.l := random( TL, 1 )
    else
        if TD < 0 then
            colour.l := 0
        else
            colour.l := random( 0, TD )

    colour.h := op2.h
    colour.s := op2.s

Output: colour ( of type colortype )
```

6.3.7 ClassDynamicalColourSystem

ClassDynamicalColourSystem provides each window with its own dynamical system which it uses to apply constraints to its window colours. As mentioned earlier, the problem of finding a global set of colours is decomposed into finding sets of colours for each window. The global dynamical system can be broken down similarly. Each window has a set of forces acting on its colours and maintains its own dynamical system to apply those forces to its colours. The dynamical system knows about four types of data; the local **ColourSet**, the local constraints, the **ColourSet**s from other windows and the constraints between other windows and the local window.

When the system is initialized, it knows about the local **ColourSet** and the default functional constraints between these colours. These include:

- a **ClassCCContrast** constraint between the border foreground and background colours.
- a **ClassCCContrast** constraint between the client foreground and background colours.
- a **ClassCCDistance** constraint between the border and client background colours ensuring they are distinguishable.

More local constraints can be added with the **addlocalconstraint** method. Conversely, local constraints can be removed with the **removelocalconstraint** method. When a local constraint is applied, the local **ColourSet** is specified for both operands.

External windows are added to the dynamical system by specifying their **ColourSet** and a single constraint. External windows remain known to the dynamical system until their constraint has been satisfied. External constraints are invoked by specifying the local **ColourSet** for the first operand and the external **ColourSet** as the second operand.

A constraint can be either a single instance of one of the constraint classes or an array of instances. To perform an action on an array of constraints, the action is performed on each array element and the result is the logical *and* of the individual results. In this way, many constraints can define the relationship between two windows, for example, but they can be managed much more simply by **ClassDynamicalColourSystem** because they are grouped together. To handle this, the **apply** method of a constraint is not involked directly, rather a cover function called **applyconstraint** is involked. This function is defined as follows:

```
Input: ColourSet1, ColourSet2, Constraint

    if type( Constraint ) = array then
        satisfied := true
        for each constraint C in array Constraint do
            satisfied := satisfied and
                         apply( ColourSet1, ColourSet2, C )
        return satisfied
    else
        return apply( ColourSet1, ColourSet2, Constraint )

Output: boolean ( indicating if the constraint is satisfied )
```

The dynamical system has its own internal "clock" which it uses to update its state. Each time the clock advances one unit, the state of the system is updated by applying all of the local and external constraints. The clock is actually implemented using NeWS events. A **TickTockDS** event is defined and **ClassDynamicalColourSystem** expresses an interest in it. If the system has not reached a state of equilibrium as a result of the current update, the dynamic system distributes a **TickTockDS** event to itself which will arrive a fixed amount of time in the future. When it receives the event, the process is repeated.

To determine if the dynamical system has reached a state of equilibrium, instances of **ClassColour** returns a boolean value when their **applyforce** method is invoked, indicating if the net force exerted on them is approximately zero. A zero net force indicates the colour is relatively stable. If all of the colours indicate that they are stable, the system is assumed to be stable and no **TickTockDS** event is sent out.

The pseudocode for the routine which updates the dynamical system is:

```
loop for ever
    wait for a TickTockDS event

    % apply the local constraints
    for i = 1 to size( LocalCons ) do
        applyconstraint( Local ColourSet,
            LocalCons[i].op2,
            LocalCons[i].constraint )

    % apply the conflicting window constraints
    for i = 1 to size( ConflictingCons ) do
        satisfied := applyconstraint( Local ColourSet,
            ConflictingCons[i].otherColourSet,
            ConflictingCons[i].constraint )

        if satisfied then
            remove constraint i from ConflictingCons

    % now, update the colours.
    allstable := true
    for each colour C in Local ColourSet do
        if iscolourobject( C ) then
            stable := applyforce( C )
            if not stable then
                newcolourgoal( C, colour(C) )
            allstable := allstable and stable

    if not allstable then
        send a new TickTockDS event
endloop
```

Once the system is stable, the only way it can become unstable again is if one of its local colours is changed manually, or if another window's colours change to conflict with it. If either of these events occur, the **nudgesystem** method is called. If a **TickTockDS** event has been sent out and the system is waiting for it, this method does nothing. Otherwise, a new **TickTockDS** event is broadcast.

The time delay of the **TickTockDS** event is proportional to the age of the window. When the window has just opened, its colours are required to change very quickly so that other colours are disturbed as little as possible. This *initial state* of rapid change lasts a relatively short time but has the desirable effect of allowing a new window to quickly move its colours out of conflict with older windows, if at all possible. A second benefit of having the time delay proportional to the age of the window is that the delay will be slightly different for each window. This avoids a regular pattern of dynamical system updates for any subset of windows, which in turn results in a much more consistent computational load on the window system because all of the windows do not update simultaneously.

6.3.8 ClassWindowStyle

This implementation of *window styles* (see Section 5.4.2) represents a very simple approach which illustrates the potential of window styles. **ClassWindowStyle** manages the organization and retrieval of the window styles. Windows are organized based on their application name and a second optional name. This optional name is ideally specified by the user to provide additional semantic information about the window, such as its purpose. However, this name must be provided by the application, something that is not supported in existing NeWS programs. Therefore, if the optional name is not defined, the hostname on which the application is running is used as a

Key	Constraint	Defining Characteristics
BBG	ClassCCVariation:	Operand1 = BBG Operand2 = HLS Colour (0, .5, 0) Relative Hue Range = $[0, 0]$ Relative Saturation Range = $[0, 0]$ Relative Perceived Br. Range = $[-.25, +.25]$
BFG	colortype	RGB Colour (0, 0, 0)
CBG	ClassCCVariation:	Operand1 = CBG Operand2 = BBG Relative Hue Range = $[0, 0]$ Relative Saturation Range = $[0, 0]$ Relative Perceived Br. Range = $[+.4, +.4]$
CFG	colortype	HLS Colour (0, 0, 0)
Extra	null	

Table 6.1: The Default Window Style Template.

default. The *window name* is the concatenation of these two names. **ClassWindowStyle** provides a method to change both the application and optional names of a window.

To allow different window styles to be created, this class defines a global dictionary called the **StylesDict** which contains all of the style templates defined in the system. A style template is a set of constraints associated with a name in this dictionary. It is called a style template, and not a window style, because a window style can be created from multiple style templates, as discussed below. Each style template is implemented as a dictionary which can contain one or more of these five keys: **BBG**, **BFG**, **CBG**, **CFG** and **Extra**. The purpose of these specific keys is explained below.

When a window is opened, a window style is created for it. Its window style is a dictionary that will contain the constraints which define its style, and is created as follows. First, the constraints for the default window style are added to the dictionary. This defines a bland, monochromatic window style, as shown in Table 6.1. Next, the application name, the optional name and the window name are looked up in the **StylesDict**. If any of these are found, their constraints are added to the window style dictionary. Thus, any keys that are defined in these style templates will replace the current keys in the dictionary.

If none of these names are found in the **StylesDict**, the default BBG constraint is replaced by a similar **ClassCCVariation** constraint which uses one of the eight basic colours described in Section 5.4.4 as its second operand. The actual colour value used for operand two is one of the mean centroid values found by Boynton and Olson (Boynton and Olson, 1987), shown in Table 6.2.

Windows with the same name are defined to be *similar*, those with different names are defined to be *different*. Therefore, all windows with the same name are assigned the same basic colour. The first time a window name is encountered, a basic colour is assigned to it at random from the set of basic colours that have not yet been assigned to any name. When the eight basic colours have been assigned, they are reused. The window difference constraints should take care of making the windows appear different, even if they are based on the same basic colour.

Once the window style has been created, the initial colours for the window are selected by invoking the **pickrandom** method of the constraints whose keys correspond to the symbolic names of the four window colours: **BBG**, **BBG**, **CFG** and **CBG**. The colours are picked in such an order that the colour a constraint depends on is

Basic Colour Name	Mean OSA Coordinates			Nearest OSA Coordinates			XYZ Coordinates		
	L	j	g	L	j	g	X	Y	Z
Red	-3.6	1.5	-6.9	-4	2	-6	14.77	10.28	6.01
Green	-0.4	3.9	2.5	0	4	2	26.56	32.05	18.68
Orange	-0.2	5.7	-6.3	0	6	-6	36.08	28.26	9.20
Purple	-2.5	-2.5	-1.8	-3	-3	-1	15.20	14.21	24.81
Pink	0.5	0.3	-4.9	1	1	-5	40.38	33.93	30.37
Brown	-3.2	2.8	-2.9	-3	3	-3	15.20	14.21	24.81
Yellow	2.6	8.2	-1.7	3	9	-1	53.10	53.86	14.24
Blue	-0.8	-2.5	2.8	-1	-3	3	19.62	23.81	39.52

Table 6.2: Centroid Values for Eight Basic Colours.

Parameter	Similar Window Value	Different Window Value
Attract?	true	false
Distance	15	80

Table 6.3: The Window Difference Constraints.

picked before that constraint is used to select a random colour. The same approach is used by the **picknewcolours** method, which is invoked to select a new set of colours for the window.

Once the initial colour values for the window are selected, the window style constraints are added to the dynamical system as local constraints. The **Extra** key in the style dictionary holds an array of additional style constraints that are not involved in the initial colour selection, but should be added to the dynamical system. Since the dynamical system is started whenever a new constraint is added, these constraints will be applied to the initial colours even though they were ignored during the initial colour selection.

ClassWindowStyle handles the differentiation between *different* and *similar* windows. As mentioned above, if windows have the same name, they are similar, otherwise that are considered different. **ClassCCWindowDistance** constraints are created to group similar windows and distinguish different windows, as shown in Table 6.3. The distances suggested in Section 6.3.6.3 for grouping similar colours and distinguishing different colours are used for the distance parameter.

A method called **checkwindow** is provided to check if an external window's colours conflict with the local colours. This method uses the window names to determine if the windows are similar or different, and invokes the **check** method of the appropriate **ClassCCWindowDistance** constraint described above to check for conflicts.

A method called **addwindow** is provided to add an external window to the dynamical system. As above, the name is used to determine if the window is similar or different and the **check** method of the appropriate **ClassCCWindowDistance** constraint is invoked to check for conflicts. If the external window's colours conflict with local colours, the external window is added to the dynamical system.

6.3.9 ClassColourWindow

ClassColourWindow is used to integrate the colour window classes into the TNT2.0 class hierarchy. To do this, it inherits from both the TNT2.0 **ClassWindow** and from **ClassWindowStyle**. This class is also responsible for the interwindow communication of the jostling constraint solver. Furthermore, the **ClassDynamicalColourSystem** introduced the concept of an *initial period* of rapid change. This concept is further supported by this class.

Window communication is simple. When a window is created it expresses an interest in an event called a **Jostle** event. Jostle events contain all of the pertinent information about a window. It then notifies existing windows of its initial colours by sending out a **Jostle** event. When a window receives a **Jostle** event, it checks to see if any of the colours in the jostling window conflict with its colours by invoking the **checkwindow** method of **ClassWindowStyle**.

If the colours conflict, one of two things happens. If the conflicting window is in its *initial state* and the local window is not, a **Jostle** event is immediately sent directly to the conflicting window and is *not* added to the local dynamical system. This informs the conflicting window of the local windows colours, allowing it to attempt to solve the conflict before its initial period is over. If the conflicting window in not in its initial state, or the local window is in its initial state, the conflicting window is added to the local window's dynamical system by invoking the **addwindow** method of **ClassWindowStyle**.

A window participates in the jostling process until it is destroyed, even when it is not visible. The decision to include a window in the jostling process when it is iconified or invisible is based on the desire to create as stable a colour environment as possible. If windows cease to jostle when they are closed (but not destroyed), colours may be required to change whenever a window is opened, regardless of how long it has been since a window was created or destroyed. These changes are avoided by having all windows continually participate in the jostling process.

Chapter 7

Conclusions and Future work

7.1 Conclusions

The aim of this thesis is to demonstrate the feasibility of a dynamic window system with colour constraints to supports aesthetic colour selection. This has been accomplished. Relevant research from many unrelated fields was reviewed, providing a firm basis on which to build the system. The important factors that must be taken into account when designing and implementing a dynamic window system with colour constraints were explored in depth and a system designed and implemented which embodied the results. The implementation shows that such systems are possible to create.

There are several key features that make the implementation work. Obviously, the most important aspect is that the colour relationships of interest can be expressed as constraints between the colours. The distributed-jostling model allows the colour selection problem to be decomposed into several smaller problems, allowing for an efficient implementation. The concept of the dynamical colour system allows the constraints to be modelled as forces between the colours, providing a simple and elegant constraint solver.

It is apparent from using the window system that much fine tuning is needed, both of the implementation and of the theoretical basic for the constraints. An ongoing period of constraint adjustment is underway in which specific values used in the implementation, such as the strengths of the forces that are used to satisfy the constraints, the time step used in the dynamical system, the mass of the colours, etc., are being adjusted.

Of greater importance, however, are the theoretical constraints, not the implementation of them. All of the constraints, especially the contrast constraint and the window separation constraints, where created using experimental results from other fields. While those results were the best guidelines available, now that the system has been shown to be feasible, these results need to be examined more carefully in the context of computer window systems.

- The contrast metric is based on experiments where reading thresholds were of interest. As a result, declines in reading speeds of up to half were considered insignificant. However, these drops in reading speed most likely resulted in a significant decline in reading comfort. Informal experiments with members of the Computer Graphics Laboratory confirmed that contrast levels much higher than those discusses by Boynton and Knoblauch are needed for comfortable use. Indeed, these researchers never claimed that their results were valid for this particular application.
- The distance metric is based on the Euclidean distance in CIELUV space. While CIELUV is perceptually uniform, it is geared toward smaller colour differences. Furthermore, no uniform colour spaces are well

suited for judging colour similarity. Two shades of a single colour may appear related in some abstract way, yet be far apart in these spaces because the colours are quantitatively different. Furthermore, the distance thresholds used were based on experimental evaluation of the segregation abilities of the basic colours. Those experiments did not focus on creating a lower bound for good colour segregation.

For each of these problems, experiments need to be designed to determine more precise constraints for use in dynamic window systems. This implementation can serve as an ideal testbed for the results of such experiments.

Another example of experimentation that can be supported with this system involves experiments aimed at discovering a more comprehensive model of window difference and similarity. A general model of how people judge coloured objects to be different or similar does not exist, yet is needed to create more useful constraints for window organization. Furthermore, more than just the difference or similarity of the component colours is involved in the grouping and segregation of windows. For example, use of the system has shown that very different window styles can create visually unrelated windows with quite similar colours. Improving techniques for window organization is important, as recent experiments (Cowan, Jolicoeur and Loop, 1991) show convincingly that colour is far superior to any other aspect of computer windows for context resolution.

Furthermore, use of the window system has suggested a need for more specialized constraints. For example, colour combinations such as fully saturated red with fully saturated blue should be avoided. While the red/blue problem is well known, other colours exhibited problems which need to be explored. Very saturated purple, for example, caused much the same problems when used with almost any colour.

The performance of the system was surprisingly good, considering the amount of processing being done in an interpreted language. However, the performance is not adequate for a production system, or even one that will be used for experimentation. It is hoped that further refinement of the constraints will alleviate the performance problems. However, if all else fails, much of the system can be recoded in C and executed as an external process. This will definitely solve the performance problems, although some flexibility for experimentation may be lost.

The implementation is based on the HLS colour space, which is not necessarily the best choice. HLS was used primarily because it is easy to visualize and explain to non-colour theorists and because is is mathematically uniform, making the implementation of most of the constraints very simple. However, this colour space is perceptually non-uniform and is not well suited for some constraints, most notably the distance constraint. A better choice would be one of the uniform colour spaces, such as CIELUV. CIELUV supports the notions of hue, lightness and saturation which are needed for the **ClassCCVariation** constraint as follows (CIE, 1978):

$$h_{uv} = \text{arctan}(v^*/u^*) \tag{7.1}$$

$$C^*_{uv} = (u^{*2} + v^{*2})^{1/2} \tag{7.2}$$

$$s_{uv} = C^*_{uv}/L^* \tag{7.3}$$

where L^*, u^* and v^* are the CIELUV coordinates or a colour value, L^* is the lightness of the colour, h_{uv} is the hue angle of the colour, C^*_{uv} is the chroma of the colour and s_{uv} is the saturation of the colour. Furthermore, L^* is a direct function of Y as follows

$$L^* = 116(Y/Y_n)^{1/3} - 16,\ Y/Y_n > 0.01 \tag{7.4}$$

(where Y_n is the value of Y for the white object-colour), making CIELUV appropriate for calculating **ClassCC-Contrast** as well.

7.2 Future Work

Much work remains to be done before this system could be interpretted as a fully usable commercial system. Most obviously, a well designed graphical interface should be created to allow users easy modification of various

aspects of the system, such as adjusting individual colour values and creating and modifying aesthetic constraints and window styles. No interactive support is currently provided for these tasks.

User criticisms are not currently handled by the system, although support for doing so exists in the form a colour mass. Colours that a user has expressed preference in merely need to have there mass increased to prevent them from moving. When a user indicates his or her likes or dislikes, they should be retained, analyzed and used to influence future choices. However, techniques for doing this effectively and efficiently need to be explored carefully, so that the ability of the system to suggest new colour combinations is not compromised. A simple way of handling this is to occasionally create some windows based on past user choices and to create others without regard for those choices. An exciting possibility for creating windows based on past user choices is found in (Salomon and Chen, 1989), where neural nets were used to generate colour sets. The generated sets have the property that they are quite similar to the seed set, which is exactly what is required here.

Another area for improvement is the handling of more than the four basic colours for a specific window. Fortunately, the dynamic window system was designed with this in mind, so adding support for more colours, and for constraints between them, would be trivial. For example, since the colours are stored in PostScript dictionaries, adding support for additional colours only requires that a consistent naming scheme be developed for these colours. The dynamical system and the constraints would automatically handle the new colours, while the window style object would need to be generalized.

Given the ability to use an arbitrary number of colours in a single window, it would be possible to automatically generate multiple background and foreground colours for a single window. For example, an application could request a background colour and any number of foreground colours for use with it. The techniques used for the generation of distinguishable window colours, as well as research such as (Corte, 1986) and (Grosse, 1985), would provide a basis for this functionality. By giving applications the ability to specify the foreground/background relationship between any pair of colours, the implementation would be straightforward.

Additional constraints for to assist with aesthetic colour selection should be developed. It is mentioned in Section 2.3 that one of the design goals of the OSA space was the capability of suggesting interesting colour harmonies. The art of Karl Gerstner (1986) demonstrates that this goal has been achieved. Lai (1991) demonstrates one possible intuitive approach to navigating the OSA colour space. This interface could be used to allow users to specify interesting aesthetic constraints based on the planes of the OSA lattice.

One class of windows that have seemingly been ignored throughout this thesis are those which contain realistic and/or complicated interiors. Handling such windows is difficult and was deemed clearly beyond the scope of this thesis from the start. The major problem with such windows is that there is not clear way to select colours that will harmonize with them, so another approach is called for. Consider the way pictures are framed. No matter how complicated the picture, the use of a sufficiently thick matte will allow it to be placed on a wall of a completely different colour or with a complicated pattern. The matte sets the picture apart from the rest of the wall, just as a thicker, neutrally coloured border would set a complicated window apart from the rest of the window system and prevent it from ruining the harmony of the window system. However, a computer screen can afford the loss of real estate represented by a thick matte far less than a wall. Other techniques will need to be investigated. In the very least, experiments should be performed to discover the minimum reasonable thickness that a border should be to provide good separation.

Another problem is that as colour becomes easier to use, the limits of colour to organize windows are discovered. Colour spaces have only three degrees of freedom, and organizing many unrelated windows in a useful fashion is quite difficult. To alleviate this problem, the colour vocabulary needs to be enriched. The obvious dimension to explore is *visual texture*. Currently, windows all have a plastic appearance, regardless of their colour. Imagine, instead, windows that appeared as marble, wood, metal, fur or water. By adding additional degrees of freedom to window appearance, especially ones that are so completely different from the current ones, the ability to organize windows is significantly increased. While there are many different ways texture could be added, and

many different textures that could be added, experiments should be performed to discover what sort of textures would be useful to support context resolution and window organization, and not merely gratuitous.

Appendix A

Relating CIELUV Units to OSA Units

Boynton and Smallman (1990) demonstrated that the *basic colours* have good segregational abilities. Part of that ability is due to their good separation in OSA space; the pairs of basic colours are on average just over ten OSA units appart. Distances as small as half of that also showed good separation is certain cases.

The adjacent colours in the OSA space are similar enough to appear related, yet clearly distinct when viewed in close proximity. Thus, they provide a good metric for the degree of difference that should be used for *similar* colours.

Unfortunately, the OSA space is not practical for computer graphics work because it only encompasses a subset of the CRT colour gamut. Worse, the defining formulae do not extrapolate well beyond the confines of the OSA space, becoming very badly behaved even within the confines of the CRT gamut. Another uniform colour space, such as the CIELUV colour space described in Section 2.2.2.2, is more appropriate for use with CRTs.

What is needed, then, are reasonable CIELUV equivalents to the OSA distances mentioned above. Since the OSA space is discrete, with only 424 colours defined, it is fairly simple to compute the distance between every pair and compare it to the distance between the CIELUV representations of the two colours. The CIELUV coordinates of each colour were obtained by converting from the XYZ coordinates found in (Wyszecki and Stiles, 1982).

Tables A.1 and A.2 show the results of the comparison. For all the pairs of OSA colours with the same interpoint distance, the average, maximum, minimum and standard deviation for the equivalent CIELUV distances are shown. The results are fairly good, showing that if only rough distance measures are required, the CIELUV space will serve the purpose. By examining the table, 80 CIELUV units is a reasonable value to represent a distance of ten OSA units. Similarly, 40 CIELUV units shall be used to represent five OSA units and 15 CIELUV units to represent 2 OSA units.

OSA		CIELUV Distance				OSA		CIELUV Distance			
Dist	Cnt	Av	Max	Min	Std.Dev	Dist	Cnt	Av	Max	Min	Std.Dev
1.7	1372	13.4	17.9	7.4	0.14	9.5	1549	77.9	102.4	51.4	0.16
2.0	992	15.3	21.0	8.4	0.22	9.8	720	78.8	100.2	56.0	0.13
2.8	1739	21.8	30.7	14.7	0.15	9.9	2035	80.0	104.7	54.5	0.14
3.3	3238	25.6	36.9	15.1	0.18	10.0	818	81.5	106.5	53.3	0.15
3.5	1078	26.9	35.2	16.7	0.13	10.2	1833	83.2	108.1	54.7	0.15
4.0	734	31.0	41.9	19.3	0.22	10.3	1725	83.9	110.6	60.2	0.14
4.4	2730	33.8	47.3	23.0	0.14	10.4	791	85.4	109.2	55.8	0.14
4.5	2640	34.7	49.3	22.7	0.18	10.7	999	87.4	111.9	62.7	0.14
4.9	2481	38.2	53.2	23.8	0.15	10.8	1474	87.3	114.9	60.6	0.13
5.2	3158	40.6	56.5	25.7	0.18	11.0	920	91.0	115.8	61.5	0.14
5.7	1062	44.1	60.8	31.4	0.15	11.1	892	91.3	115.4	59.3	0.13
5.9	4007	46.2	65.3	31.2	0.16	11.3	191	92.9	117.9	73.9	0.15
6.0	2518	47.0	63.6	30.4	0.15	11.4	1840	95.3	121.0	62.6	0.13
6.3	1842	49.8	69.0	32.4	0.19	11.5	720	94.6	118.5	66.6	0.14
6.6	1789	51.5	68.2	35.0	0.13	11.7	664	95.0	123.2	70.7	0.13
6.6	1739	52.3	71.8	33.9	0.17	11.8	936	97.6	122.2	65.6	0.13
6.9	567	54.4	68.5	39.3	0.12	11.8	611	98.8	123.7	71.1	0.13
7.1	3096	56.5	76.5	36.6	0.17	12.0	361	98.3	123.3	65.1	0.13
7.2	1488	56.8	78.2	39.1	0.16	12.1	614	102.1	127.7	70.7	0.12
7.5	2810	59.0	80.8	41.1	0.14	12.2	275	105.8	127.5	66.3	0.10
7.7	4018	60.8	83.8	41.0	0.15	12.3	719	103.7	127.7	67.1	0.12
8.0	324	64.9	82.8	41.7	0.19	12.4	893	102.8	128.7	73.9	0.12
8.2	1184	65.2	86.1	44.3	0.15	12.6	206	110.1	133.1	75.4	0.09
8.2	2339	65.8	88.5	43.7	0.15	12.8	195	107.6	131.1	86.1	0.15
8.5	1619	68.0	90.5	44.6	0.16	12.8	731	108.3	133.2	76.0	0.12
8.7	2372	68.8	93.4	50.3	0.15	13.0	345	109.0	133.0	82.8	0.14
8.7	1006	69.0	91.7	53.7	0.14	13.1	821	111.8	135.7	80.5	0.12
8.9	925	71.9	96.5	49.5	0.16	13.1	151	106.6	128.2	81.1	0.12
9.1	2709	73.3	96.1	47.8	0.15	13.3	142	115.3	135.0	90.6	0.08
9.2	1741	73.7	98.3	50.7	0.15	13.4	656	114.1	139.3	84.3	0.11
9.4	838	74.2	93.9	57.5	0.12	13.4	391	112.7	139.8	84.7	0.13

Table A.1: A comparison of OSA and CIELUV distances.

OSA		CIELUV Distance				OSA		CIELUV Distance			
Dist	Cnt	Av	Max	Min	Std. Dev	Dist	Cnt	Av	Max	Min	Std. Dev
13.6	240	117.9	139.7	85.6	0.10	16.6	47	142.1	164.5	117.4	0.08
13.7	217	115.7	140.0	87.6	0.11	16.7	9	139.0	151.8	108.1	0.13
13.9	24	106.4	121.6	91.7	0.11	16.8	22	144.5	167.1	125.4	0.08
14.0	373	119.5	144.7	86.8	0.10	17.0	12	156.5	166.7	147.7	0.05
14.0	204	120.5	142.9	93.7	0.09	17.1	18	153.7	167.5	120.6	0.12
14.1	268	118.8	146.7	89.6	0.12	17.1	12	156.5	167.8	138.6	0.08
14.2	322	124.4	145.4	96.6	0.11	17.2	34	147.6	169.5	126.3	0.08
14.3	161	124.8	146.2	95.6	0.11	17.3	32	151.4	168.7	123.6	0.07
14.4	78	126.7	147.7	97.7	0.12	17.3	16	147.8	169.4	123.8	0.13
14.5	170	121.7	145.9	94.5	0.14	17.4	4	161.9	163.9	160.7	0.01
14.6	196	127.6	151.0	96.9	0.10	17.5	18	146.8	171.3	130.5	0.10
14.7	226	126.4	150.4	96.0	0.10	17.5	8	155.7	169.7	148.9	0.04
14.8	208	127.5	151.2	96.4	0.10	17.7	6	149.0	169.1	121.5	0.13
15.0	91	127.6	149.5	99.7	0.11	17.7	18	149.2	170.2	128.5	0.08
15.1	201	129.9	152.0	96.8	0.10	17.9	4	148.1	172.2	139.6	0.11
15.1	58	127.3	148.9	98.2	0.10	18.0	7	154.9	160.5	151.1	0.02
15.2	44	133.8	156.1	120.1	0.08	18.0	9	147.0	166.9	137.1	0.06
15.3	70	135.8	157.2	123.7	0.06	18.1	1	157.2	157.2	157.2	NaN
15.4	99	132.9	155.6	99.4	0.10	18.2	3	153.2	164.3	146.4	0.06
15.6	116	134.8	155.4	104.5	0.10	18.2	2	160.6	162.1	159.2	0.01
15.6	74	134.3	156.7	102.7	0.12	18.3	4	154.0	167.3	148.2	0.06
15.7	104	138.7	157.6	108.8	0.10	18.4	3	153.0	161.6	146.6	0.05
15.8	144	138.7	161.4	101.7	0.11	18.4	1	154.7	154.7	154.7	NaN
16.0	2	140.3	144.6	135.9	0.04	18.5	1	159.3	159.3	159.3	NaN
16.1	79	139.8	161.8	111.6	0.09	18.6	3	142.8	155.3	134.8	0.08
16.1	68	140.9	162.3	107.4	0.10	18.8	1	162.1	162.1	162.1	NaN
16.2	52	141.2	161.9	119.3	0.10	18.9	2	149.3	156.0	142.6	0.06
16.3	20	141.4	152.9	108.6	0.09	19.0	3	147.4	156.9	139.3	0.06
16.4	15	139.5	157.3	113.7	0.08	19.1	1	161.1	161.1	161.1	NaN
16.5	16	142.9	164.2	136.3	0.05	19.3	2	149.7	151.4	147.9	0.02
16.6	59	142.5	166.3	111.3	0.09	19.5	1	155.1	155.1	155.1	NaN

Table A.2: A comparison of OSA and CIELUV distances.

Appendix B

Object Oriented Programming

A good overview of object oriented programming principles and terminology may be found in (Wegner, 1987) or (Meyer, 1988). Details of the object oriented extension to PostScript used in NeWS can be found in (Sun, 1990). This appendix provides a list of the key object oriented terms needed to understand the thesis.

The following list of terms is taken from (Wegner, 1987).

- **object:** An object has a set of "operations" and a "state" that remembers the effect of operations. The value returned by an operation on an object, unlike typical functions, may depend on its state as well as its arguments.
- **object-oriented language:** A language that supports objects which belong to classes and has class hierarchies which may be incrementally defined by inheritance. In other words:

  object-oriented = object + class + inheritance

- **class:** A class is a template from which objects are created. Objects of the same class have common operations. Classes have an interface that specify the operations accessible to clients. The class specifies code for implementing operations in the class interface.
- **inheritance:** A class may inherit operations from "superclasses" and may have its operations inherited by "subclasses." An object of class X may use the operations defined by class X and its superclasses. Inheritance from a single superclass is called *single inheritance*; inheritance from multiple superclasses is called *multiple inheritance*.

The following additional definitions are taken from (Sun, 1990).

- **instance:** An instance is one of the objects described by a class; an instance inherits its variables and procedures from its class.
- **instance variables:** Instance variables are given to each instance of a class.
- **class variables:** Class variables are shared by all instances of a class. Together with the instance variables, they define the "state" of an object.
- **methods:** Methods are procedures that a class uses to operate on its instances. A *message* is sent to an object to invoke a method. Methods are the "operations" that effect the "state" of an object.

Appendix C

Constraint Classes

Two constraints, **ClassCCAnalogousVariation** and **ClassCCWindowDistance** were created to illustrate how simple it is to create new constraints by subclassing from the three basic constraints. They are included here to illustrate that point.

```
%%
%% A useful variation on ClassCCVariation constrains
%% colours to be analogous to other colours.
%%
/ClassCCAnalogousVariation ClassColourConstraint
dictbegin
    % style is one of; -1, 0 or 1
    % There are two analogous colour schemes for any given
    % colour, one to the left and one to the right.
    %     1 means use both
    %    -1 means the colour is in the right sector
    %     0 means it is in the left sector
    /AnalogousMethod 0   def

    % A monitor is needed because the instance vars are
    % changed and the constraint could be used multiple
    % times.  Things could get nasty otherwise.
    /MyMonitor null def

dictend
classbegin

    /NewInit {
        % initialize our parent
        /NewInit super send

        % a monitor is needed to synchronize access
        /MyMonitor createmonitor promote

        % Hue mode must be /Absolute
        /HueMode /Absolute promote
    } def

    % disable setting hue mode, since it must be absolute.
    /sethuemode { % mode => -
        pop
    } def
```

```
% allow the analogous method to be changed.
/setanalogousmethod { % method => -
    /AnalogousMethod exch def
} def

% do a (mod 12) to keep the hues in the range of 0..11
% PostScript's mod operator allows negative results!
/modhue { % huesector => validhuesector
    12 mod
    dup 0 lt {12 add} if
} def

/setanalogoushuerange { % hue => -
    AnalogousMethod                 % hue meth
    exch 12 mul                     % meth 0..12
    dup 12 eq {pop 0} if cvi        % meth sec

    % is it "random" or one of the two methods?
    1 index 1 eq {
        % random range goes from sec-1 to sec+2
        exch pop                    % sec
        dup 1 sub exch              % sec-1 sec
        2 add                       % sec-1 sec+2
    } {
        % range goes from meth+sec to meth+sec+2
        add dup 2 add               % s+m s+m+2
    } ifelse

    % fix the hues.
    modhue 12 div exch modhue 12 div exch

    % set the parents hue range to the allowed hues.
    sethuerange
} def

% check:  return true if the constraint is satisfied,
%         false otherwise
/check { % op1 op2 => bool
    MyMonitor {
        dup getoperand2 getoperandraw pop pop
        setanalogoushuerange
        /check super send
    } monitor
} def

% apply:  return true if the constraint is satisfied,
%         false and a force vector otherwise
/apply { % op1 op2 => bool
    MyMonitor {
        dup getoperand2 getoperandraw pop pop
        setanalogoushuerange
        /apply super send
    } monitor
} def

% select a valid random colour
/pickrandom { % op1 op2 => -
    MyMonitor {
        dup getoperand2 getoperandraw pop pop
        setanalogoushuerange
        /pickrandom super send
```

```
        } monitor
    } def
classend def


%%
%% the difference between two windows is slightly more
%% complicated than just the distances between the
%% component colours.
%%
/ClassCCWindowDistance ClassCCDistance [/MyMonitor]
classbegin

    /NewInit {
        /NewInit super send
        % a monitor is needed to synchronize multiple
        % access
        /MyMonitor createmonitor def
    } def

    % the defining colour is the one with the highest
    % chroma.  We only consider BBG and CBG.
    /getchroma { % l s => c
        % get the distance l is from .5
        exch .5 sub abs

        % reverse and double it. .5 => 1, 0,1 => 0
        .5 exch sub 2 mul

        % scale the saturation
        mul
    } def

    /getdefiningcolour { % CS1 CS2 => /label true |
                         %            false
        .1 null [ /BBG /CBG ] {         % cs cs max lab nl
            4 index 1 index get         % ... op1
            4 index 2 index get         % ... op1 op2
            getoperandhls
            getchroma                   % ... op1 h2 c2
            exch pop                    % ... op1 c2

            exch getoperandhls
            getchroma
            exch pop                    % ... c2 c1
            max                         % ... c'
            dup 4 index ge {
                4 -1 roll pop 3 1 roll  % cs cs max lab nl
                exch pop
            } {
                pop pop
            } ifelse
        } forall                        % cs cs max lab

        4 1 roll pop pop pop

        % if none of the colours have a high chroma,
        % trivially accept the windows since trying to
        % move low chroma colours causes the system to
        % misbehave.
        dup null eq {pop false} {true} ifelse
    } def
```

```
    % check:  return true if the constraint is satisfied
    %         false otherwise
    /check { % op1 op2 => bool
        MyMonitor {
            2 copy getdefiningcolour {  % op1 op2 dc2
                dup setoperand1         % op1 op2 dc2
                setoperand2
                /check super send
            } {                         % op1 op2
                pop pop true
            } ifelse
        } monitor
    } def

    % apply:  return true if the constraint is satisfied,
    %         false and a force vector otherwise
    /apply { % op1 op2 => bool
        MyMonitor {
            2 copy getdefiningcolour {  % op1 op2 dc2
                dup setoperand1         % op1 op2 dc2
                setoperand2
                /apply super send
            } {
                pop pop true
            } ifelse
        } monitor
    } def

    % select a valid random colour
    /pickrandom { % op1 op2 => -
        MyMonitor {
            2 copy getdefiningcolour {  % op1 op2 dc2
                dup setoperand1         % op1 op2 dc2
                setoperand2
                /pickrandom super send
            } {
                pop  pop
            } ifelse
        } monitor
    } def
classend def
```

Bibliography

Albers, J. (1971). *Interaction of Color*. Yale University Press. text of the original 1963 edition with selected color plates.

Allen, R. E., editor (1990). *The Concise Oxford Dictionary of Current English*. Oxford University Press, 8th edition.

Baecker, R. M. and Buxton, W. A. (1987). *Readings in Human–Computer Interaction: A Multidisciplinary approach*. Morgan Kaufmann Publishers, inc.

Berlin, B. and Kay, P. (1969). *Basic Color Terms*. University of California Press.

Boff, K. R., Kaufman, L., and Thomas, J. P., editors (1986). *Handbook of Perception and Human Performance*. Wiley, New York, Toronto.

Boynton, R. M. (1979). *Human Color Vision*. Holt, Rinehart and Winston.

Boynton, R. M. and Olson, C. X. (1987). Locating basic colors in the osa space. *Color Research and Application*, 12:94–105.

Boynton, R. M. and Olson, C. X. (1990). Salience of chromatic basic color terms confirmed by three measures. *Vision Research*, 30(9):1311–1317.

Chevreul, M. E. (1967). *The Principles of Harmony and the Contrast of Colors*. Reinhold, New York. Originally published in 1854.

Christ, R. (1975). Review and analysis of colour coding research for visual displays. *Human Factors*, 17(6):542–570.

CIE (1978). Recommendations on uniform color spaces – color difference equations – psychometric color terms, supplement #2 to cie publication #15. Available in the US from the National Bureau of Standards. Available in Canada from the National Research Council of Canada.

Corte, W. D. (1986). Finding appropriate colors for color displays. *Color Research and Application*, 11(1):56–61.

Cowan, W., Jolicoeur, P., and Loop, S. (1991). Experiments in context resolution comparing various aspects of computer windows. Performed recently at the University of Waterloo.

Cowan, W. B. (1989). Colorimetric properties of video displays. Notes for Course 25 at the Annual Meeting of the Optical Society of America.

Cowan, W. B. and Ware, C. (1985). Colour perception tutorial notes: Siggraph'85. Notes for Course 5.

Cowan, W. B. and Wein, M. (1990). State versus history in user interfaces. In et al., D. D., editor, *Human-Computer Interaction - INTERACT '90*, pages 555–560, North-Holland. Elsevier Science Publishers B.V.

Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F. (1990). *Computer Graphics: Principles and Practice*. The Systems Programming Series. Addison-Wesley Publishing Company, New York, 2nd edition.

Gerritsen, F. (1975). *Theory and Practice of Color*. Van Nostrand Reinhold Company, New York. Translation of Het fenomeen kleur, by Ruth de Vriendt.

Gerritsen, F. (1988). *Evolution in Color*. Schiffer Publishing Ltd., West Chester, PA. originally published in Dutch in 1982 as Evolutie van de Kleurenleer. Translation by Dr. Edward Force and Ruth de Vriendt.

Gerstner, K. (1986). *The Forms of Color*. MIT Press. Translation of: Die Formen der Farben.

Gould, J. D., Alfaro, L., Finn, R., Haupt, B., Minuto, A., and Salaun, J. (1987). Why reading was slower from crt displays than from paper. In Carroll, J. M. and Tanner, P. P., editors, *Human Factors in Computing Systems and Graphics Interface*, pages 7–11. CHI + GI.

Grosse, E. (1985). Automatic choice of colors for level plots. Technical report, AT&T Bell Laboratories. Numerical Analysis Manuscript 85-1.

Hall, R. (1988). *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, New York.

Helson, Judd, and Warren (1952). Object-color changes from daylight to incandescent filament illumination. *Illuminating Engineering*, 47:221.

Hope, A. and Walch, M. (1990). *The Color Compendium*. Van Nostrand Reinhold, New York.

Itten, J. (1961). *The Art of Color: The subjective experience and objective rationale of color*. Van Nostrand Reinhold Company, New York. Translated by Ernst van Haagen.

Klassen, R. V. (1989). *Device Independent Image Construction for Computer Graphics*. PhD thesis, University of Waterloo. Available as Research Report CS-91-19 from the Department of Computer Science.

Knoblauch, K. and Arditi, A. (1989). Effects of character size and chromatic contrast on reading performance. *Applied Vision: 1989 Technical Digest Series*, 16:98–101.

Knoblauch, K., Arditi, A., and Szlyk, J. (1990). Effects of chromatic and luminance contrast on reading. In press: Journal of the Optical Society of America A.

Lai, J. (1991). Implementation of colour design tools using the osa uniform colour system. Master's thesis, University of Waterloo.

Legge, G. E. (1989). Reading: Effects of contrast and spatial frequency. In *Applied Vision: 1989 Technical Digest Series*, volume 16, pages 90–93.

Legge, G. E., Parish, D. H., Luebker, A., and Wurm, L. H. (1990). Psychophysics of reading. xi—comparing color contrast and luminance contrast. *Journal of the Optical Society of America*, 7(10):2002–2010.

Legge, G. E., Rubin, G. S., and Luebker, A. (1987). Psychophysics of reading—v. the role of contrast in normal vision. *Vision Research*, 27(7):1165–1177.

MacAdam, D. L. (1974). Uniform color scales. *Journal of the Optical Society of America*, 64(12):1691–1702.

Meier, B. (1987). Effective use of color in user-computer interface design: Final report. Technical report, Brown University, Department of Computer Science, Box 1910, Providence, RI 02912.

Meier, B. J. (1988). Ace: A color expert system for user interface design. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, pages 117–128.

Meyer, B. (1988). *Object-oriented Software Construction*. Prentice Hall International Series in Computer Science. Prentice-Hall.

Munsell, A. H. (1947). *A Color Notation: An Illustrated System Defining All Colors and Their Relations be Measured Scales of Hue, Value and Chrome*. Munsell Color Company, Baltimore, MD.

Norman, R. B. (1990). *Electronic Color: the art of color applied to graphic computing*. Van Nostrand Reinhold.

Ostwald, W. (1931). *Color Science: A Handbook for Advanced Students in Schools, Colleges, and the Various Arts, Crafts, and Industries Depending on the Use of Color*. Winsor and Newton.

Png, T. (1991). Stack windows. Technical report, University of Waterloo.

Quiller, S. (1989). *Color Choices*. Watson-Guptill Publications, 1515 Broadway, New York, N.Y. 10036.

Rogers, D. F. (1985). *Procedural Elements for Computer Graphics*. McGraw-Hill.

Rubin, G. S. and Legge, G. E. (1989). Psychophysics of reading. vi—the role of contrast in low vision. *Vision Research*, 29(1):79–91.

Salomon, G. and Chen, J. (1989). Using neural nets to aid color selection. Preprint.

Schlueter, K. G. (1990). Perceptual synchronization in window systems. Master's thesis, University of Waterloo. Available as Research Report CS-90-36 from the Department of Computer Science.

Seim, T. and Valberg, A. (1986). Towards a uniform color space: A better forumla to describe the Munsell and OSA color scales. *Color Research and Application*, 11:11–24.

Smallman, H. S. and Boynton, R. M. (1990). Segregation of basic colors in an information display. *Journal of the Optical Society of America*, 7(10):1985–1994.

Sun (1990). *NeWS 2.1 Programmer's Guide*. Sun Microsystems, Inc. Part Number: 800-4888-10.

Sun (1991). *The NeWS Toolkit Reference Manual*. Sun Microsystems, Inc. Part Number: 800-5543-10.

Tanenbaum, A. S. (1986). *Operating Systems: Design and Implementation*. Prentice Hall Software Series. Prentice-Hall, Inc.

Tinker, M. A. (1963). *Legibility of Print*. Iowa State University Press.

Uchikawa, H., Uchikawa, K., and Boynton, R. M. (1989). Influence of achromatic surrounds on categorical perception of surface colours. *Vision Research*, 29(7):881–890.

Uchikawa, K. and Boynton, R. M. (1987). Categorical color perception of japanese observers: Comparison with that of americans. *Vision Research*, 27(10):1825–1833.

Wegner, P. (1987). Dimensions of object–based language design. *SIGPLAN Notices*, 22(12):168–182. Proceedings of the OOPSLA'87 Conference, Oct. 4–8, 1987, Orl ando, Florida.

Wyszecki, G. (1954). A regualr rhombohedral lattice sampling of munsell renotation space. *Journal of the Optical Society of America*, 44:725.

Wyszecki, G. and Stiles, W. (1982). *Color Science: Concepts and Methods, Quantitative Data and Formulae*. Wiley, New York.