A Tale of Two Toolkits: Developing an Application Using First and Second Generation Object Oriented User Interface Technology.

Imran Bashir
Department of Electrical &
Computer Engineering
111 Link Hall, Syracuse University
Syracuse, NY 13244

David M Sternlicht
MIT X Consortium
545 Technology Square
Cambridge, MA 02139

November 12, 1992

Abstract

This paper takes an objective look at developing an application using the Intrinsic based Athena widgets, and the C++ based InterViews toolkit. The test application chosen is a color editor that interprets and sets stimuli values using the X color management system (Xcms). A view data paradigm was chosen to organize the relationship between the color displayed, and its representation in the various color spaces.

Electronic Address: ibashir@cat.syr.edu & dave@expo.lcs.mit.edu

Contents

1 Introduction

While they are orthogonal mechanisms, the use of color and the use of gui toolkits are important tools through which modern programmers design their applications. Xcms, distributed with release 5 of X11, has provided programmers with a more generic way of specifying colors and guarantying their consistency across different display terminals.

InterViews, soon to be called Fresco, has provided a new, and perhaps a more powerful and flexible, means of designing and implementing a user interface.

We have blended these two important systems to create a tool called, ICE, an InterViews-based Color Editor. ICE represents the eight Xcms color spaces. Each color space is represented by a set of scroll bars, one per dimension of the color space. These dimensions are individually changeable. The effect of a change in a dimension of a color space is reflected in the dimensions of the other color spaces. Each color space is itself selectable in that it updates its values with a change in a value in some other color space. There is also a color-viewing area that displays the color for the given values of RGB on a display. At any point in time, the values of all color spaces are equivalent and correspond to the color that is displayed on the screen.

1.1 Goals

The main goal of this project was to build an application using the InterViews toolkit. This would help determine the leverage InterViews can provide in building an application. The application that we chose is a multi-space color editor. The two other intended benefits of this work are:

1. A color editor that represents all the eight color spaces.
2. A comparison of InterViews toolkit with X Athena Widgets.

1.2 Constraints

The following requirements and constraints were placed on the design in order to enhance the tool's usability.

1. The tool should represent all the eight Xcms color spaces, namely, RGB, RGBi, CIEXYZ, CIExyY, CIEu'v'Y, CIEL*u*v*, CIELab, TekHVC.
2. The value of any dimension in each of the above color spaces should be individually changeable.
3. A change in one dimension of a color space should be reflected in all the other color spaces. Thus a change in the value of H in TekHVC color space should update the values of all the dimensions in the other color spaces.
4. There should be a color viewing area that represents the color on that CRT for the current values of all color spaces.
5. A user should be able to enable(disable) any color space that she is interested(not interested) in.

6. The design of the tool should be such that it is easy to add a new color space. In other words, the design should be enhanceable.

This paper is organized as follows. In the next section, we give a brief history of color control on computer displays. A brief description of each of the eight color spaces is also given. After that the design of the ColorEditor tool using InterViews, called ICE hereafter, is explained in detail. A second design using Athena Widgets is also given. The problems encountered during the design of this tool, especially in using InterViews and libX11 are detailed in the following section. This is followed by a comparison of designs using InterViews and Athena Widgets. The paper concludes with a discussion of what we have learned from the design of this tool. Some recommendations for InterViews are also given.

2 Background

In the early days of computer graphics, there were only monochrome displays and each pixel on the display could be controlled by turning it on or off. An improvement over these displays were the gray level displays. In these display units, a user could specify several levels of intensity for a single pixel. The biggest problem with these displays was the non-linear relationship between the DAC(Digital-to-Analog Converter) gun value and the amount of light emitted from the screen. A myriad of correction algorithms, to make this relationship linear, were suggested in subsequent years [4].

The advent of color displays provided a major break-through in graphics technology. The early color displays used the same model for displays as their monochrome counterparts. A color display had three electron guns(Red, Green, and Blue). The color of each pixel was controlled by turning each of the three guns on or off. Thus a user could specify eight different colors using these three electron guns. The model that describes this color space is known as RGB and is still in widespread use. The RGB model uses a Cartesian coordinate system that describes a color space within the confines of a cube. The combination of all three colors produces white and absence of all the three produces a black color. An enhancement of this model was to use different levels of intensities for each primary color. Thus, for example, a 0-255 RGB system can produce over 16 million unique combinations of RGB intensities. This model again had the same problem of a nonlinear relationship between DAC gun value and the emitted light from the screen. Therefore it is hard for a user to envision a color by selecting a triplet of values. For some values of RGB, a small change may not effect the color apparently while the same amount of change may effect the color appreciably for some other values of RGB.

RGB values are device dependent; meaning that for the same values of RGB, the color produced by one display may very well be different from another display [3]. This poses a hurdle for software vendors as a combination of colors on one display may look good while the same color specifications may produce an ugly effect on some other

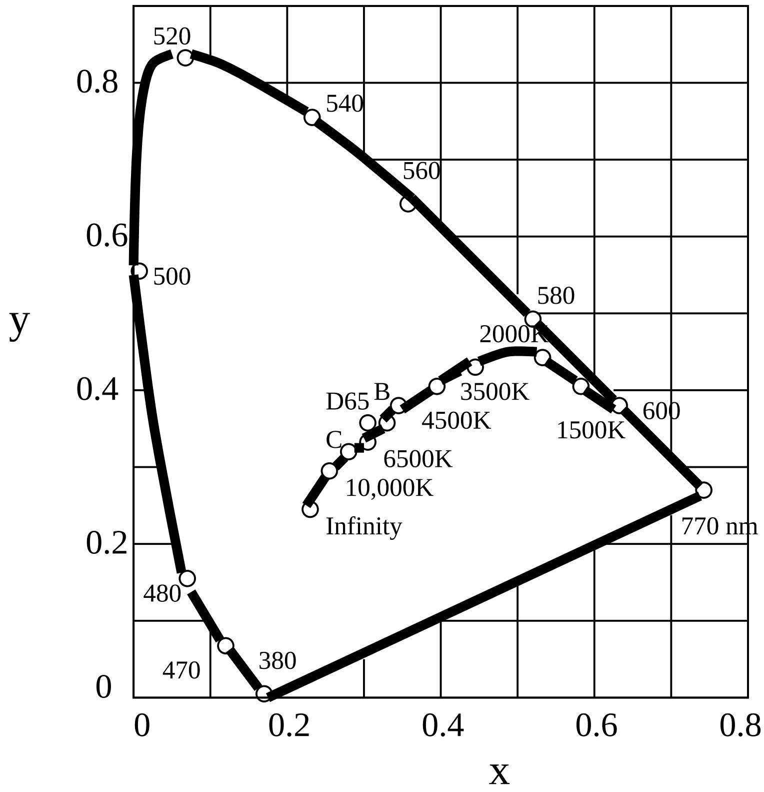

Figure 1: The 1931 Chromaticity Diagram

display.

This motivated the need for *device-independent* color spaces. By specifying a color in a device independent color space, the same color would be displayed on two different screens.

The basis of device-independent color spaces is the CIE colorimetric system. The CIE (*Commission Internationale de l'Eclairage*) or the *International Commission on Illumination* colorimetric system defines the essential standards, mathematical definitions, and measurement procedures for the industry. This system is based on the premise that a color stimulus results from the proper combination of an illuminant, or light source, an object, and an observer [5].

The 1931 CIE Chromaticity Diagram, shown in figure 1 [1] was the first diagram accepted internationally for general use [5]. All of CIE's color spaces are based on this diagram.

In 1991, Tektronix provided the X Color Management System to the X Consortium for incorporation into its fifth release. RGB and CIE, along with the new TekHVC system, were included and are described in the next two sections.

[1]This diagram is a hand-drawn copy of the original and may not be exact.

2.1 Device Dependent Color Spaces

There are two device dependent color spaces, RGB and RGBi.

RGB: This is a device dependent additive color space. Colors are formed by the combination of the three primary colors, Red, Green, and Blue. The different levels for each of these primary colors can produce a wide range of colors. The benefit of this color space is its ease of implementation. The disadvantages of this color space are its device dependency and the difficulty of perception by a user.

RGBi: The RGBi color space is essentially the same as RGB. This color space is *gamma-corrected* which means that the intensity levels can be spaced logrithmatically rather than linearly, to achieve equal steps in brightness.

2.2 Device Independent Color Spaces

These days multi-color displays are being used in almost every walk of life. Some of these uses are highly sophisticated and there is a serious need to accurately control and specify the color displayed by a CRT. Following is description of each of the device independent color space models available today.

CIExyY: The 1931 Chromaticity Diagram describes the xyY color space. Here x and y are the chromaticity coordinates and contains no information on the inherent lightness of a color. The tristimulus value Y represents the luminance factor. This color space is the basis of all the device independent color spaces. Its greatest advantage is its value in predicting the additive mixture of two or more colors. The disadvantages of this color space are two fold. One, it does not have a visually equally spaced colors. Second, the black color has no unique position in this color space [5].

CIEXYZ: In 1931, CIE defined three new standard primaries, namely X, Y, and Z, to replace the R,G, and B primaries. These new primaries can be used to match, with only positive weights, all the visible colors. The 1931 Chromaticity Diagram was obtained by plotting x and y for all visible colors, which is the projection onto the (X, Y) plane of the (X + Y + Z = 1) plane of this model [1].

The CIEXYZ color space also serves as the "hub"for the conversions between device-independent and device-dependent color spaces [4].

CIEu'v'Y: This color space is the latest implementation of a linear transformation of the 1931 Chromaticity Diagram. The objective of this linearization was to have a color space in which colors were distributed in a more uniform manner. The u' and v' are the chromaticity coordinates where Y represents the luminance factor.

CIEL*u*v*: The efforts to linearize the relationship between the magnitude of a color difference and its appearance continued. The scientists combined the u,v Diagram with the 1976 metric lightness function L* which resulted in L*u*v* color space. This color space is based on the opponent-color theory which models the human color vision. In this color space, u* and v* describe the chromatic attributes of color. The u* represents red-green coordinate, while the v* axis represents yellow-blue. The L* axis describes variations in lightness. This color space has a unique position for the black color. Today CIEL*u*v* is considered as the best available approximation to a visually uniform color space [5].

CIEL*a*b*: L*a*b* color space is colorimetric and is very close to being perceptually uniform [7]. It has separate lightness and chroma channels. It is represented as a Euclidean space with the three quantities L*, a*, and b* plotted along axes right angles to each other. L* represents lightness, a* represents the red/green axis, and b* represents the yellow/blue axis. This color space is extremely useful for device-independent manipulation of continuous tone images [7].

TekHVC: The TekHVC color space, developed at Tektronix, is a set of algorithms that creates a three-dimensional color space [6] [4]. H(ue) describes the property associated with the color family. V(alue) denotes the variation in lightness. C(hroma) denotes the saturation or vibrancy of a color. HVC provides a color space that is visually uniform [5].

2.3 InterViews

InterViews is an object-oriented software system for window-based applications. It is an C++ interface to X windows; it is built on top of X. It was implemented at Stanford University by Mark Linton's group. The name "InterViews" comes from the idea of a user interface object presenting an *Inter*active *view* of the same data [2]. InterViews provides a number of high level components for building interfaces.

At present, only 3.1Beta version of InterViews is available with the promise of a complete 3.1 version sometimes in near future. Obviously this version is not stable yet.

3 Design of ICE

3.1 Implementation using InterViews

The design of the `ice` tool has primarily been driven by the philosophy of data versus views. This dichotomy has a long history in the user interface community. Smalltalk divided information display into a Model-View-Controler triad [15]. In the Andrew toolkit, some of the most basic components are data objects and views [16]. InterViews follows in this tradition by also promoting the concept of different views of the same data.

The mechanism for supporting this consists of three protocols: *Adjustable, Observable,* and *Observer* [2]. Adjustables handle requests to modify a viewing area in ways such as scrolling, paging, and zooming. An Observable takes care of a single dimension of an Adjustable and represents InterViews notion of data. Observables communicate

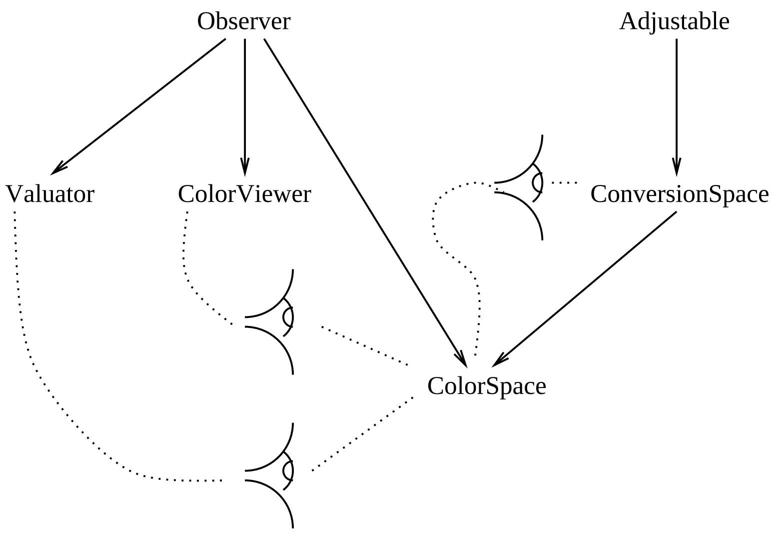

Figure 2: Hierarchy of classes showing Observer/Observable relationship

with a number of views, known as Observers in the InterViews toolkit. This is done through a notify-update protocol.

We have used these classes in designing our color editor. The hierarchy of classes is shown in figure 2 while the design of the tool is shown in figure 3.

The hierarchy figure indicates that ConversionSpace is an Adjustable while the ColorSpace is an Adjustable as well as an Observer. As an Observer, a ColorSpace observes a ConversionSpace. Through the Observables maintained by being an Adjustable, the ColorSpace lets both the ColorViewer and the Valuator objects observe itself. ColorViewer is an Observer and is responsible for updating the color viewing area on the screen. One Valuator per dimension of a ColorSpace keeps a FieldEditor up to date and also lets a user change the value of that dimension through the FieldEditor.

In our design, ConversionSpace serves as common data while the ColorSpace objects acts as several views of this data. Each ColorSpace uses the ConversionSpace object to convert between its local values (view) and the global values (common data). Every ColorSpace has three scroll bars to visually represent each color dimension. Three FieldEditors (subclassed from InputHandler) per ColorSpace textually show the values in each dimension of a ColorSpace. Therefor a value in a particular dimension can be changed by either moving the thumb of the corresponding scroll_bar or by entering a number in the FieldEditor through the keyboard.

The ColorViewer object is responsible for displaying the color and is therefor an Observer of the RGB ColorSpace. When there is a change in the values of this Col-

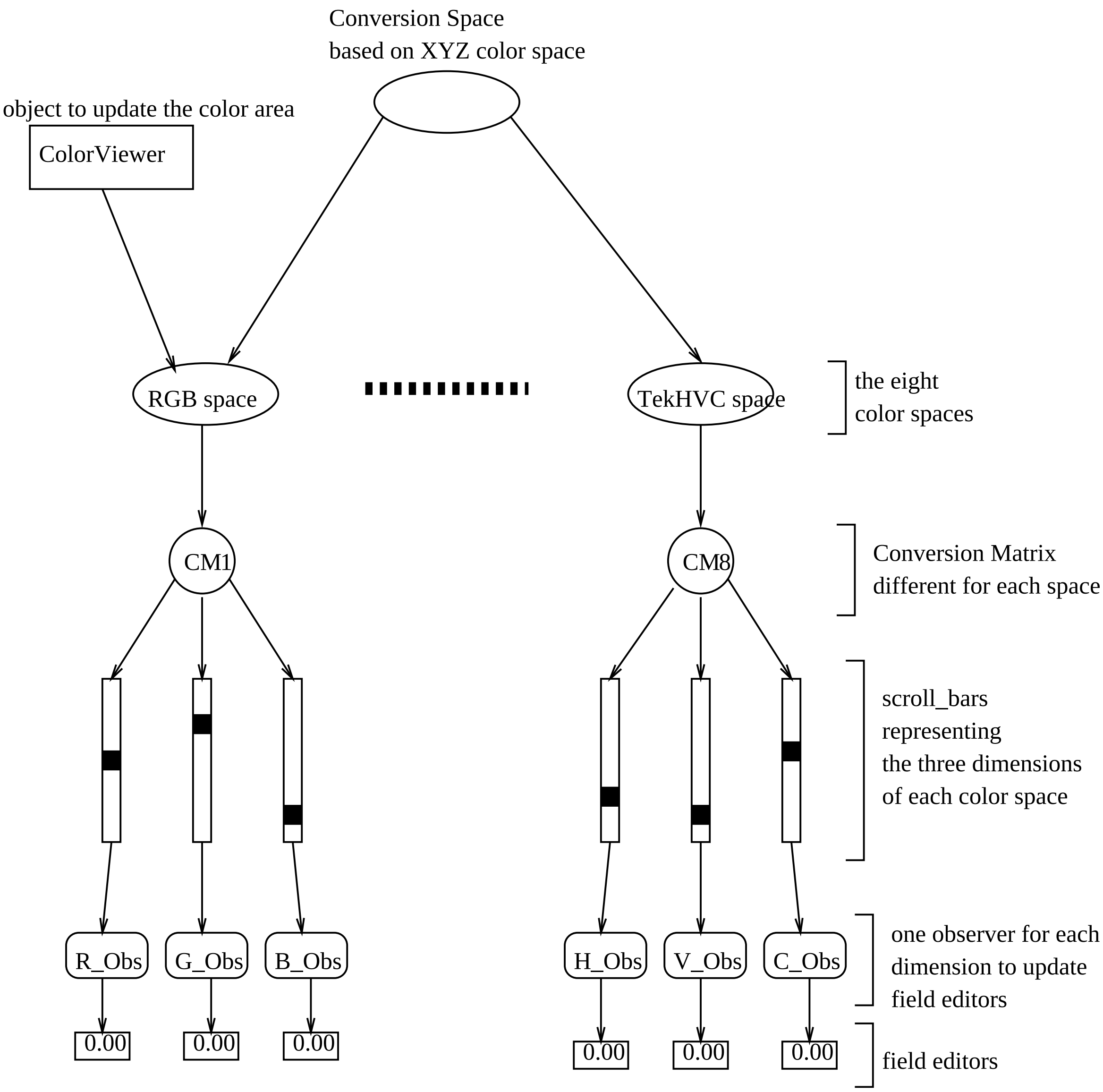

Figure 3: Design of the ColorEditor tool ICE

orSpace, the ColorViewer is notified and it updates the ColorViewer's viewing area on the screen.

Dynamic execution of the **ice** program would proceed as follows:

1. Initialization: When a ConversionSpace is created, the values in the three dimensions of the ConversionSpace are each set to 0.5. When a ColorSpace is created, it calculates its local values by observing the ConversionSpace's values of X, Y, and Z, and then updating itself.
2. Manipulation: When a user changes a value in a ColorSpace, either by disturbing the position of a scroll_bar or through a field editor, the following takes place:
   - A change in value calls the ColorSpace's scroll_to() method. This method changes local color values to new values, then calls the routine `convertToXYZ()`, which changes XYZ values in the ConversionSpace object.
   - All the ColorSpaces are observers of the ConversionSpace object. As soon as a change is observed, a ColorSpace's update method is invoked.
   - `ColorSpace::update()` calculates its local values by looking at the new values of X,Y, and Z in the ConversionSpace. Once the new values have been determined, local copies are updated.
   - This setting of local values also notifies any other Observers, namely Valuators or the ColorViewer, of this ColorSpace object.

A potential problem here is the possibility of an infinite loop. Consider the following scenario: A ColorSpace object, CS1, has a change in one of its dimensions. This change in CS1 will also alter the values of the ConversionSpace object. A change in ConversionSpace alerts all its Observers to update themselves. Since CS1 is also an Observer of the ConversionSpace object, it will try to update itself without realizing that it was the one that had changed the ConversionSpace object to begin with. A simple boolean data member `iSetConversionflag` is added to ColorSpace to indicate an object is the initiator of a color change and that its values do not have to be updated with the change of X, Y, and Z in the ConversionSpace object.

3.2 Implementation using Xaw

The Athena widget implementation seeks to provide the extensibility provided by the *Observable/Observer* paradigm supported in Interviews. The latter implementation uses a single observable to maintain a color data state, and various observers to provide a view of this data. The Xaw implementation subclasses a color management object (CMSObject) from the Object class (a superclass of Core). This object provides resources as follows:

1. An Xcms structure containing CIEXYZ data. This is our "data".
2. A list of callbacks where each callback performs a translation to a given color space. This is our set of "views".

Every set of color space controls can be activated or deactivated by a toggle switch that adds or removes a callback from the CMSObject callback list. The design is extensible because to add a conversion to another code space one simply needs to add a callback to the list.

4 Problems Encountered

4.1 InterViews

InterViews, as mentioned earlier, is still in the process of development and a final version of 3.1 has not been released yet. We used the InterViews 3.1Beta version for `ice`. Since this is the beta release, some problems were expected of this package.

The major problem encountered with InterViews 3.1Beta [2] was the handling of multiple dimensions. Let us first describe the problem in detail.

4.1.1 Problem

The previous chapter described the 3 data view protocols: *observable, observer,* and *adjustable*. The declaration of dimensions in `iv/src/include/InterViews/geometry.h` looks like this:

```
typedef unsigned int DimensionName;

enum{
   Dimension_X = 0, Dimension_Y, Dimension_Z, Dimension_Undefined
};

class CoordinateSpace {
public:
   enum { x = 0, y, z, dimensions };
};
```

In the standalone `enum` above, four types of dimensions have been defined, i.e, Dimension_X, Dimension_Y, Dimension_Z and Dimension_Undefined. This approach has two problems. One problem is with the Adjustable class itself and the second problem is the representation of these dimensions through the *WidgetKit* class interface. Let's look at both these problems individually.

Adjustable Problem An *Adjustable* object should be able to handle n dimensions. In an application where a user has a three-dimensional space, an *Adjustable* object can

[2] We shall refer to this as IV3.1 for the rest of this paper.

be used as such. Each dimension in the application's 3-D space can be represented by the dimensions in the stand-alone `enum` above.

Assume that we have an n-dimensional space, where n = 4. We create a subclass of the Adjustable base class and call it `FDAdjustable` (for Four-Dimensional-Adjustable). We can declare the following to have the fourth dimension for this declared class:

```
enum { Dimension_W = 5}; // including Dimension_Undefined

class FDAdjustable : public Adjustable {
...
```

In this subclass, we store all the information about the Dimension_W, such as the current value in the Dimension_W etc. Now if we create an object of the type FDAdjustable and try to *attach* an Observer to the Dimension_W, we create a serious programming error [3].

To comprehend why this happens, let's look at the constructor of our base class Adjustable.

```
struct AdjustableImpl {
private:
    friend class Adjustable;

    AdjustableInfo info_[CoordinateSpace::dimensions];
};

Adjustable::Adjustable() {
    impl_ = new AdjustableImpl;
    AdjustableImpl& a = *impl_;
    for (DimensionName d = 0; d < CoordinateSpace::dimensions;
              d++) {
        AdjustableInfo& i = a.info_[d];
        i.observable_ = new Observable;
        i.small_ = 1;
        i.large_ = 0;
    }
}
```

As is seen in the first code fragment of this chapter, the value of `CoordinateSpace::dimensions` is 3, and the index of `AdjustableInfo info_` ranges from 0 to 2. Therefor, following a natural design impulse to add dimensions is bound to create problems.

In conclusion, one can only have three dimensions in an Adjustable and any space with 4 or more dimensions has to be represented by more than one Adjustable.

[3] Namely a segmentation fault

WidgetKit class Problem We have seen how the Adjustable class fails to handle more than three dimensions. Let's now turn our attention to representing these dimensions in real world.

The current value of a variable in a particular dimension can graphically be represented by a scroll_bar (or a slider). We shall only discuss scroll_bars; the same discussion can be extended to sliders as well.

The current interface of the WidgetKit class for scroll_bars looks like this:

```
class WidgetKit {
  public:
    ...
    virtual Glyph* hscroll_bar(Adjustable*) const;
    ...
    virtual Glyph* vscroll_bar(Adjustable*) const;
    ...
    virtual Glyph* scroll_bar_look(DimensionName,
                                   Adjustable*) const = 0;
    ...
};
```

In this interface *hscroll_bar* always uses the Dimension_X of the Adjustable and returns a horizontal scroll_bar. Similarly *vscroll_bar* always uses the Dimension_Y of the Adjustable and returns a vertical scroll_bar. The only method in which one can specify the DimensionName for an Adjustable is *scroll_bar_look*. But if we look at the implementation of this method, it only handles the cases of Dimension_X and Dimension_Y, for which it returns a horizontal and vertical scroll_bar respectively.

This clearly indicates our limitations. With the current interface, the following problems are clearly visible:

1. No dimension other than the Dimension_X and Dimension_Y can be graphically represented by a scroll_bar.
2. One cannot represent Dimension_X of an Adjustable through a vertical scroll_bar. Similarly Dimension_Y cannot be represented by a horizontal scroll_bar. This functionality might be needed where one has more than two dimensions for an Adjustable and one needs to represent all these dimensions horizontally or vertically.

4.1.2 Proposed Solution

Having identified two problems that don't allow the handling of multiple dimensions with an Adjustable, we now propose some solutions:

Solution to Adjustable class problem The most obvious solution in this case is that the client of an Adjustable should be able to specify the number of dimensions that it wants for a given adjustable. Thus there should be another Adjustable constructor that takes the number of dimensions n, as an argument. Depending on this value of n, `AdjustableInfo info_` array should be created.

Solution to WidgetKit class problem The WidgetKit class interface should be general enough to handle the cases mentioned in the problems section. Thus the following extension to the interface of the WidgetKit class is proposed.

```
class WidgetKit {
  public:
    virtual Glyph* hscroll_bar(Adjustable*,
                               DimensionName d=Dimension_X) const;
    virtual Glyph* vscroll_bar(Adjustable*,
                               DimensionName d=Dimension_Y) const;

    virtual Glyph* scroll_bar_look(DimensionName,
                                   Adjustable*) const = 0;
    virtual Glyph* hscroll_bar_look(DimensionName,
                                   Adjustable*) const = 0;
    virtual Glyph* vscroll_bar_look(DimensionName,
                                   Adjustable*) const = 0;

    ...
};
```

In this interface for a *hscroll_bar*, one can also specify the dimension of the given Adjustable for which a horizontal scroll_bar is required. Similarly a vertical scroll_bar can be created for any dimension of the Adjustable by using the *vscroll_bar* interface. If no dimension is specified for these methods, they behave just like the original *hscroll_bar* and *vscroll_bar* methods.

Two new methods: *hscroll_bar_look* and *vscroll_bar_look* have been added so that a horizontal or vertical scroll_bar can be created for any dimension of an Adjustable.

The implementation of these methods require some other changes in the interface of some other classes. We shall look at these changes now.

4.1.3 Changes Required

WidgetKit interface Other than the changes mentioned above, the method

virtual Stepper up_mover(Adjustable*) const;*

should be extended to

virtual Stepper up_mover(Adjustable*, DimensionName d=Dimension_Y) const;*

This makes sure that a stepper at the end of a scroll_bar can be used for any dimension for which its scroll_bar area is being used.

Similarly *down_mover, left_mover and right_mover* are also extended. The default value of DimensionName for up_mover and down_mover is Dimension_Y and for left_mover and right_mover is Dimension_X.

Slider interface For the implementation of horizontal and vertical scroll_bars, IV3.1 declares two subclasses of Slider class. These subclasses are class XSlider and YSlider. The XSlider creates a horizontal slider area in the scroll_bar for Dimension_X while the YSlider provides vertical slider area in the scroll_bar for Dimension_Y of the given Adjustable.

This limits the representation of dimensions of an Adjustable to only two dimensions, namely Dimension_X and Dimension_Y. To ensure that other dimensions can also be represented by scroll_bars and that any dimension can be represented by a vertical or horizontal scroll_bar, we create two subclasses of Slider class. These subclasses are known as HSlider and VSlider. The constructors of these classes are given below:

```
HSlider(Style*, Adjustable*, DimensionName);
VSlider(Style*, Adjustable*, DimensionName);
```

The protocol of *Slider::allot_thumb_major_axis* should be extended so that it takes an additional argument. This last argument specifies the Horizontal or Vertical Direction along which the major axis of the given slider is oriented. If no Direction is specified, then an Undefined Direction is used and the method works just like the original one.

4.2 libX11

The creation of this tool was also obstructed by some entities other than IV3.1. The most notable of these is libX11.

4.2.1 Problem

After having implemented our design, we observed that the response of `ice` was very slow. After running it through the *profiler*, we found the routines `_XcmsSquareRoot` and `_XcmsArcTangent` to be the culprits. Between these two routines, sometimes the execution time was as high as 80%. These routines are called from the color-conversion routines of the libX11. Discussion with Bob Scheifler, about this issue, revealed that libX11 does not use the *math* library routines for functions like cosine, sine, tangent etc. Instead, it provides its own implementation of these functions. These implementation are much slower than their *math* library counterparts.

4.2.2 Solution

We provided our own implementation of the two routines `_XcmsSquareRoot` and `_XcmsCubeRoot`. We created a new file `XcmsMath.c` and instead of the implementation provided by the libX11 version of these routines, we invoked the corresponding *math* library routines. For example, our implementation of `_XcmsSquareRoot` looks like this.

```
double _XcmsSquareRoot(a)
double a;
{
    return sqrt(a);
}
```

We also provided the *math* library implementation of the routines `_XcmsCosine`, `_XcmsSine`, and `_XcmsArcTangent` in the file `XcmsTrig.c`. Two object files were created for these two C files and these object files were replaced in the libX11.a. This library was then used to create the executable for `ice`.

The response of our tool improved appreciably with this solution.

5 InterViews versus Athena Widgets

5.1 A Basis Of Comparison

One of the central goals of this work is to gain insight into the strengths and weaknesses of Xt and the InterViews toolkits. This is a difficult but interesting proposition because the systems are so different. Why do we compare InterViews with Xt/Athena and not, say, Xt/Motif? While it is true that Motif contains more functionality than Athena, the authors felt that the comparison should center more upon the underlying systems than on GUIs. Also we wanted to limit our consideration to freely available software because improvements attributable to the financial resources commercial establishments possess were not considered germane to the base technologies' essential worth. Our task here is to find some common criteria for comparison. Both systems show a number of strengths and weaknesses. We have subdivided the areas of comparison into eight sections: Architecture, Rendering, Portability and Code Size, Subclassing and Inheritance, Geometry Management, Event Management, and finally Resource Management.

5.2 Architecture

Both systems draw a line between their internal components and the components one sees on the screen, but they do so in different places. This can be seen in figure 4.

The mechanism/policy organizational dichotomy of Xt is well known. InterViews divides its core functionally from look and feel issues by separating the world into the

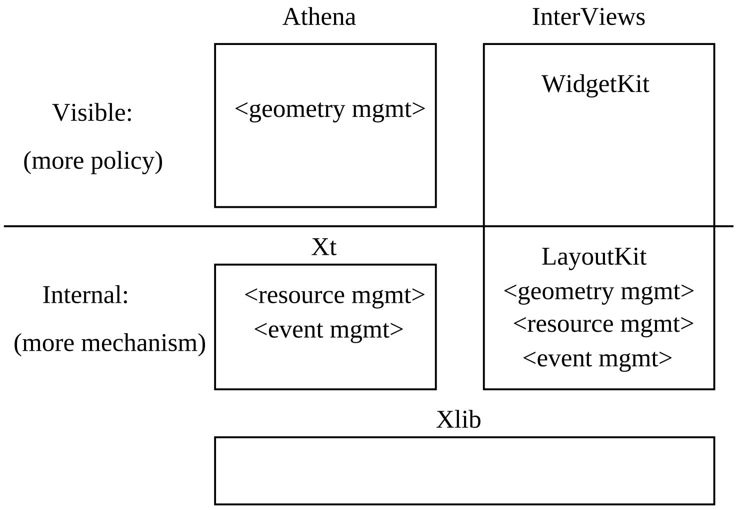

Figure 4: Architecture of Xt and InterViews

glyph hierarchy and kits. Kits are class factories that produce visible components. The largest architectural difference between the two systems is that in Xt, the composite hierarchy resides mostly above the line, or in the widget library whereas in InterViews, composite management takes place entirely below the line in the TEX abstractions of the layout kit.

5.3 Rendering

Both toolkits use a different target drawable for their base components at the Xlib layer. Xt widgets generally render to an X window whereas InterViews glyphs render to an X window Region. [4] Not maintaining a window per object allows InterViews to support many more objects than are used in a typical Xt application. The best example of this is the document editing program **doc**. By using a glyph per character, documents of over 185 thousand separate C++ objects can be easily supported [13]. InterViews and Xt also differ in that whereas widgets typically render directly to an X window, InterViews glyphs render to an intermediate object known as a Canvas. Canvas is an object that supports a structured graphics protocol similar to the PostScript language.

[4] I am intentionally not considering gadgets. They were added to Xt as an afterthought. I am also not considering InterViews components that contain X windows, namely Interactor and GLContext. These are exceptions to the rule.

InterViews also supports a subclass of Canvas that supports hardcopy output. There is no equivalent to this in Xt.

5.4 Portability and Code Size

Interviews and Xt applications build on numerous platforms. However InterViews maintains a porting layer between window systems not present in Xt. Providing this division of code helps to ensure that any investment in application code will be preserved when we move on the next window system, whatever that may be.

Certainly the size of an applications source code has something to say about the complexity of expression in the toolkit. `ace` consists of 844 lines of source code [5]. `ice` is comprised of 1561 lines. These numbers would appear to be in Xt's favor. The `ice` programs design should probibley be restudied with an eye towards reducing the number of its abstractions, and the code to implement it.

Shared libraries have made executable size no longer as pressing an issue. However until memory is free we will have to live with some constraints. On a SUN4 processor running SunOS 4.1.2, `ace` took up 32768 bytes. `ice` consumed 90112. Again these numbers favor Xt although I expect over time continual improvement of c++ compilers will narrow this.

5.5 Subclassing And Inheritance

This is the area where Xt is most severely deficient. The lack of object oriented facilities in C leaves the task of implementing subclasses in the hands of the component designer. This can be a very tedious and error prone process. Consider the following two code fragments, comparing subclassing in Xt and InterViews/C++.
C++:

```
class Foo : public  SuperFoo (, MultipleInheritanceSuperFoo) {
  // lots of stuff
};
```

Xt:

```
typedef struct _FooClassPart {
  int ignore;
} FooClassPart;

typedef struct _FooClassRec {
  CoreClassPart core_class;
    ...
```

[5]The result of adding up the .h, .c, and .ad files.

```
  SuperFooClassPart super_foo_class;
  FooClassPart foo_class;
} FooClassRec;

typedef struct FooPart {
  /* lots of stuff */
} FooPart;

typedef struct _FooRec {
  CorePart core;
    ...
  SuperFooPart superfoo;
  FooPart foo;
} FooRec;
```

To perform a single subclassing operation in Xt one must declare 4 C types. By comparison, using InterViews one must only be concerned with a single C++ class declaration.

While the mechanism for subclassing in Xt is more complex, the options available to the programmer who wishes to use its inheritance are very limited. To inherit a method, the widget designer must provide a special constant in the class field for that method. This is usually `_XtInherit` cast to an appropriate type. These kinds of details are hidden from the C++ class designer by the compiler. When one subclasses in C++, the default is for inheritance to be implicit. Xt only supports single inheritance whereas C++ provides multiple inheritance. InterViews as well as **ice** take advantage of this feature. Although multiple inheritance creates considerable complexity, it also assists in developing our applications to model the real world [8]. One of the most powerful mechanisms available to the C++ programmer is the virtual member function. These functions allow the programmer to pass a pointer to an object of a derived class into a context where only the interface of the base class is known, and expect that the derived classes member function will be executed. This is also known as polymorphism, one of the keys to programming in C++. It is much more difficult to conceive of doing something analogous in Xt. First of all, a widget programmer is not supposed to access Xt methods outside of a widget's .c file. Rather widget implementations provide external or public functions that allow applications to use methods. Indeed, there is no widget code ever written that I am aware of that passes around and manipulates a widget pointer with anything near the flexibility achieved with a C++ class pointer. While one can implement polymorphism in Xt I'm not sure you would want to. The programmer would have to be very careful. For example, one could write a function (internal to a widget's implementation of course) that accepted a generic widget pointer to a subclass of Composite and then invoked its `insert_child` method. If you were to do this you had better be sure that you have implemented this function and filled in the appropriate class part field with its address, otherwise you have committed a serious programming error. C++ by comparison, guarantees that invoking a subclasses member function would resolve to an actual function, otherwise the program simply would not compile.

Both systems differ as to when they implement their inheritance. For Xt, inheritance for a subclass occurs during runtime when the first widget of the given subclass is dynamically created. Encountering the `_XtInherit` constant tells a class to pull a method function pointer out of the appropriate field in its superclass record and place it in the subclass. This is known as *class part initialization*. By contrast, C++ defines inheritance at compilation and link time, through a number of stratagems [9]. This gives the C++ compiler the ability to check the class structure for correctness, as well as performing optimizations, such as class flattening [10].

Xt does however, have a feature not present in C++. Fields can be added to a class record without recompilation of the application through the use of an extension class record. No such mechanism exists in C++, where the addition of a class member mandates recompilation.

5.6 Geometry Management

We can divide geometry management up into three categories. Structure, or how the system organizes aggregates of components, layout, ie the initial placement of user interface objects when the application is displayed on the screen, and reconfigurability, or how the system reorganizes sub-components when one or more of them wish to change their size.

5.6.1 Structure

Creating components to physically manage sub-components is accomplished in Xt through subclassing from the Composite and Constraint widget classes. Their architecture is as follows:

Composite class:

```
typedef struct _CompositeClassRec {
    CoreClassPart  core_class;
    CompositeClassPart composite_class;
} CompositeClassRec;

typedef struct _CompositeClassPart {
  XtGeometryHandler geometry_manager;
  XtWidgetProc      change_managed;
  XtArgsProc        insert_child;
  XtWidgetProc      delete_child;
  caddr_t           extension;
} CompositeClassPart;
```

```
typedef struct _CompositeRec {
   CorePart core;
   CompositePart composite;
} CompositeRec;

typedef struct _CompositePart {
  WidgetList children; /* list of children */
  Cardinal num_children; /* number on list */
  Cardinal num_slots; /* max size of list */
  XtOrderProc insert_position;          /* method */
} CompositePart;
```

Constraint class:

```
typedef struct _ConstraintClassRec {
    CoreClassPart      core_class;
    CompositeClassPart composite_class;
    ConstrainClassPart constraint_class;
} ConstraintClassRec;


typedef struct _ConstraintClassPart {
    XtResourceList resources;   /* constraint resource list */
    Cardinal num_resources;     /* number of 'em */
    Cardinal constraint_size;   /* size of the constraint struct */
    XtInitProc initialize;      /* method */
    XtWidgetProc destroy;       /* method */
    XtSetValuesFunc set_values; /* method */
    caddr_t extension;
} ConstrantClassPart;

typedef struct _ConstraintRec {
    CorePart core;
    CompositePart composite;
    ConstraintPart constraint;
} ConstraintRec *ConstraintWidget;

typedef struct _ConstraintPart {
    int *empty;
} ConstraintPart;
```

Conceptually, composite widgets are containers that usually have been given the responsibility for maintaining their children in a uniform fashion. All widgets must ask their parents to make changes such as resize or movement. A number of widget methods accomplish this such as InsertChild, DeleteChild, Resize, QueryGeometry, GeometryManager, and ChangeManaged. Constraint widgets, by comparison are a subclass of Composite that differ in that management is based on additional information associated with each child. This information is found in a constraint structure that is pointed to by a field of the CorePart of a widgets instance record.

There are 15 Widgets in the Athena library that subclass from Composite or Composite along with Constraint. All of them are abstractions of common containers we find in the real world. They are Box, Dialog, Form, Paned, Porthole, Tree, Viewport, and various Shells. By contrast the geometry management paradigm of InterViews is far more complex. Besides simple containers, InterViews goal is to geometricly handle full text formatting functionality along the lines Of Donald Knuths $\TeX$ system[11]. As in $\TeX$, Boxes and Glue are the conceptual objects that the basic rendering object or glyph is organized by. Composite components in InterViews are created by invoking member functions of the LayoutKit class. Kits can be viewed as software factories that organize the dispensing of objects. An abbreviated list of LayoutKits composite objects include box, overlay, deck, back, front, glue, discretionary, strut, space, center, fixed, flexible, natural, margin, stretch, and span, 9 member functions, all of which heavily reflect the needs of document formatting. I would argue that the InterViews model is superior, because the page layout model subsumes the simpler model of Xt and Athena. This allows for more sophisticated schemes of geometry management.

5.6.2 Layout

Placement of objects in Xt occurs throughout the phases of widget creation, management, and finally realization. In the creation phase, after all widgets invoke their `class_initialize` methods, widgets whose parents are Constraints have a constraint record allocated and its address stored away. If a widget's parent is a composite, the intrinsics puts itself into its parent's children list by calling the parent's `insert_child` method. Management is the process by which lists of widgets are added to the geometry managed set of their parent Composite. Realization is the process of creating a widget's X window to the screen. After a series of calls to create widgets and manage subtrees, an application has a tree of unrealized widgets, some of which are managed and some of which are not. Composite realization is achieved by invoking the `change_managed` method for each child of a composite with any managed children. This is done in recursive post-order. Therefore layout starts at the leaves of the composite hierarchy and works upwards. This lets layout algorithms know what the preferred size of the children should be. Changes can ripple up and down the tree, but since a widget can not request new geometry as a result of being resized the process is guaranteed to eventually terminate. I should note here that creating but not managing widgets is a useful technique by which the Xt applications programmer can create large structures, such as menus before he really needs them, and then quickly display them on the screen. No comparable mechanism exists in Interviews.

The InterViews layout system is quite different in nature. Glyph layout and rendering takes place in 3 phases, corresponding to three member functions. I will call these the requisition, allocation, and drawing phases. These phases correspond to the 3 glyph member functions Glyph::request, Glyph::allocate, and Glyph::draw.

Requisition phase: In the first phase glyphs make their space needs known. The outermost glyph has its request member invoked, passing in a Requisition object. Requisitions objects contain information such as how much space a glyph needs, and how much they are willing to compromise by shrinking or stretching. Container glyphs call the request members of their children to compute their Requisitions, which are stored away for later.

Allocation phase: In the second phase the outermost glyph has its allocate member function invoked, passing in an Allocation object. Allocations contain the size of a glyph that is actually given. Container glyphs call the allocate members of their children and use the previously stored Requisiton information to compute their Allocation. This information is also stored away.

Drawing phase: In the third and final phase the outermost glyphs draw member is invoked, passing it a Canvas object as well as an Allocation. Container glyphs invoke the draw members of their children causing them to be drawn on the Canvas using the geometry information saved in the Allocation.

I find the InterViews system to be more symmetrical and elegant. Again C++ assists here. Geometry management is a matter of mixing functionality with information and language support of this is the implementation of the Requisition and Allocation classes is superior.

5.6.3 Reconfigurability

Both systems support the notion of adding and deleting a child from its parent aggregate. Xt defines two methods in the Composite widget class to do this, namely InsertChild and DeleteChild. InterViews accomplishes this through six member functions in the Glyph class, namely append, prepend, insert, remove, replace, and change. Again we see that the InterViews mechanism subsumes the Xt model.

Both systems provide flexibility when a child's geometry needs conflict with the parents management policy. In Xt this is accomplished through the childs invocation of the parents GeometryManger method. The parent can return XtGeometryYes in which case the child can resize. Returning XtGeomtryDone to the child means Xt has already accomplished the resize. XtGeomtryNo constitutes a denial of the request.

InterViews dictates reconfigurability through the alignment operations of the Layout kit. These operations affect a glyphs Requisition. Layout::center aligns a glyphs origin in the request/allocate process described earlier. Layout::fixed make glyphs ridged and unresizable. Layout::flexible makes them stretchable and shrinkable. Margins can also be specified. User interface objects can be managed like characters in formatting system by using the Discretionary, Composition, and Compositor classes. While all of these techniques create a flexible system for the GUI designer, the text formatter paradigm will present a certain learning curve to the vast majority of engineers who grew up using the management technique found int the in the more traditional Xt widget libraries.

5.7 Event Management

Events constitute the dynamic aspect of window systems. The X window system defines 17 basic event types but this number belies the complexity of event management. Therefore it is essential to provide good tools for organizing an applications reactions to the events delivered. Before toolkits arrived, event programming in the X window system consisted of nothing more than creating a C switch statement in the main event loop of a program. Both InterViews and Xt have the notion of an event structure. Of course Xt simply uses the `XEvent` struct, while InterViews provides a window system independent abstraction called the Event class. In Xt, one abstracts the event loop by providing a series of functions known as event handlers and registering them with the intrinsics using the `XtAddEventHandler` function. InterViews provides the Handler class to effect event management. Handlers are objects that are passed an event to process, in much the same way as Xt event handlers. However InterViews provides some interesting subclasses of Handler that provide additional functionality. InputHandlers provide the programmer with a highly configurable focus management system for mouse and keyboard input. Essentially one is provided with a tree structure of input objects and a mechanism for traversing that tree. The class protocol allows you to change focus by traversing that tree. InputHandlers take up physical space, and participate in the allocation protocol. InputHandler's share with the Patch class an ability to cause layout to change by invoking their redraw member. Indeed, InputHandler::redraw and Patch::redraw are important member functions in that they effect "damage" on the canvas where glyphs are being drawn. Damage in InterViews is the mechanism by which glyphs are redrawn. In order to redraw a glyph one damages an object known as an extension, which comprises the geometry of the glyph on the canvas. Extensions are initialized in the glyph allocation phase. Canvas damage is somewhat analogous to the the expose widget method in Xt, except it would not be unreasonable to use this technique in ones application, instead of merely refreshing a window. Indeed it is possible for no damage to occur when an obscured InterViews application is poped to the front of a window stack. Characters do not damage the canvas when they are exposed. Resizeing a application that contains a slider will cause damage to the canvas because the size of the thumb needs to change. [6]

Although InterViews presents some good ideas Xt maintains a performance edge thorough its tighter coupling with the X system. Clients can request that the server compress its motion events, generating them only when the pointer starts or stops moving. Such granularity of control would violate InterViews X independence, but results in slower programs.

Both systems provide the ability to react to file IO and to invoke functions based on the expiration of a timer. Xt does this by registering input handlers and timeout proc's

[6] The internal representation of a slider contains a thumb Patch.

with the functions `XtAppAddInputHandler` and `XtAppAddTimeOut`. Xt input handlers are specifically useful in designing graphical interfaces for standard unix utilities [12]. InterViews provides these mechanisms through the IOHandler class, namely the **inputReady**, **outputReady**, and **timerExpired** members. Xt however, provides the capability to use time that wold otherwise be lost waiting for events by registering work procedures through the `XtAppAddWorkProc` function call. No such mechanism is available in InterViews.

Another major strength of Xt over InterViews is the translation table, action procedure mechanism made available through the Intrinsic's resources. Using this one can write a number of functions that perform various jobs, and bind them at runtime to various events. As with workprocs InterViews provides no such capability.

This can be useful in designing graphical interfaces for standard unix utilities [12]. Xt provides the XtAddInput call. InterViews provides the Dispatcher class. Both systems support handling timeouts, and workprocs. InterViews lacks a notion of translation tables and accelerators. this is bad. Also InterViews has no notion of selections, although this is currently being considered.

5.8 Resource Management

Resource management is an area where Xt and InterViews differ substantially. While the InterViews paradigm is organized in a substantially different manner than Xt. I am not convinced that it is superior.

The major conceptual difference between the systems is that the Xt resource system is build around the physical organization of an application whereas InterViews has chosen to create a separate or logical hierarchy. Whereas physical structure in an InterViews application is embodied in its glyph hierarchy, logical or resource structure is managed by a separate class known as a Style.

Xt's resource mechanism has proven itself in the real world and is a major reason for the toolkits success. It starts with the Xrn calls available in Xlib, and gives the application programmer substantial functionality by simply incorporating a certain widget into his application and setting its resources. Both toolkits allow the user to provide resources on the command line and in a database file [7]. Both systems support the notion of resource converters. Xt allows resources to dynamically change through invoking a widgets **SetValues** method whereas InterViews will implement this through Styles trigger member function [8]. Despite these similarities we see substantial differences. Programmers using widgets simply add them to their program and begin to use their resources. To affect a component in InterViews, one must link the physical kit component with its logical style component in order to reach it with attributes stored in a resource file. Yes this is more flexible but it makes for more work. Indeed, InterViews resource management is quite a bit more complex than Xt's. One needs to understand

[7] Although to date, while Xt supports the *, ., and ? pattern matching constructs, InterViews only supports *

[8] This has not yet been implemented

and be comfortable with only two concepts to use Xt resources, namely **class** and **instance**. InterViews makes you become familiar with **Styles**, the distinction between **physical** and **logical hierarchies**, a styles **name**, style **prefixes**, **attributes**, and finally **aliases**. Many GUI programmers who have had great success with Xt resources are not going to be pleased with the complexity of the new system.

The designers of InterViews assert that the tight coupling between customizable attributes and the layout of objects in Xt is useless for objects such as documents where attributes apply to logical rather than physical groupings[14]. While this may be so I am not convinced yet that the document formatting paradigm is best. What other application domains would this approach provide leverage in? Monitoring nuclear power plants?, Games? Data Base Management? The abstraction seems to be overkill for 90 % of the toolkit application domain. Indeed, in the document formatting business, their are very successful products such as **FrameMaker** that use traditional Xt mechanisms with no difficulty.

Finally it is claimed by the InterViews designers that by dividing up functionality that toolkits have traditionally coupled, InterViews is able to provide a unified framework for user prefences, document attributes, and structured graphics state which should provide applications that are simpler to build, more consistent to use and easier to integrate [14]. While some good evidence has been provided for this in the case of the **doc** program, their is no major application that shows this for structured graphics [9]. Their is not enough evidence that the style methodology creates a resource design environment that is simpler and more consistent. Whether it is easier to integrate depends on what you are trying to integrate it with. Certainly not Xt programs. It would not have been difficult to incorporate the Xt mechanism into InterViews. One could have used **Xrm** to provide resource setting through a physical and logical hierarchy. Sometimes it is unnecessary to re-invent the wheel.

6 Conclusion

This paper has described the design and implementation of 2 color editors, `ace` and `ice`, written in the Athena/Xt and InterViews toolkits. At present, two versions of `ice` exist. The first version has been implemented using the changes that we have suggested in InterViews. In our local environment using SUN workstations and cfront they seem to work fine. These changes are also upward compatible, so that any code written using the old interface also works.

The second version of `ice` has been implemented using InterViews3.1 beta as it is. It and ACE will be made publicly available by anonymous ftp on export.lcs.mit.edu in the file /contrib/ACEvsICE.tar.Z.

Various aspects of the InterViews design were examined during the design of this tool. Their advantages and disadvantages were discussed between us. The changes that

[9] InterViews 3.1beta2 contains a glyph editor, but this application is just a toy.

we have proposed in this paper, were sent to Mark Linton and the handling of multiple dimensions could be supported, in the full release of InterViews3.1.

At this point, we would also like to mention that the documentation available for InterViews is still somewhat scarce. More detailed documentation should be made available. Also there are not enough examples that can be used as a model to develop custom applications. Thus, for an InterViews novice, programming in InterViews seems like an uphill battle. Hopefully, as more people adopt InterViews as their user interface library, good examples and documentation will become prolific.

References

[1] Foley,J.D., and vanDam,A., and Feiner,S.K., and Hughes,J.F., Computer Graphics Principles and Practice, Second Edition, Addison-Wesley, 1990

[2] Linton,M.A., and Calder,P.R., and Interrante,J.A., and Tang,S., and Vlissides,J.M., InterViews Reference Manual Version 3.1-Beta. June 26, 1992.

[3] Scheifler,R.W., What's New in Release 5, in *The X Resource*, Issue 0, O'Reilly & Associates,Inc. pp 7-16, Fall 1991

[4] Tabayoyon,A., and Taylor,J.M., Xcms: Integrated Color Management for X11, Tutorial Notes *Xhibition'91*, June 5, 1991.

[5] Taylor,J.M., and Tabayoyon,A., An Introduction to Device-Independent Color and Color Management, Tutorial Notes *6th X Technical COnference*, Jan 13, 1992.

[6] TekColor Color Management System, System Implementor's Manual, March 1990, Tektronix, Inc. Beaverton, Oregon.

[7] TIFF, Revision 6.0, Final Q June 3, 1992., *Aldus Developers Desk, Aldus Corporation*, Seattle, WA.

[8] Myers,S., Effective C++, Addison-Wesley, pp 157-169, 1992.

[9] Stroustrup,B., and Ellis,M., The Annotated C++ Reference Manual, Addison-Wesley, pp 217-237, 1990.

[10] Bashir, I., and Bellur, U., A Class Flattening Tool., Conference Proceedings *C++ at Work 1991*, Nov 18, 1991.

[11] Knuth, D., The Texbook., Addison-Wesley, 1984.

[12] Young, D., X Window Systems Programming And Applications With Xt, Prentice Hall, pp 142-151, 1989.

[13] Calder, P., and Linton, M. The Object-Oriented Implementation of a Document Editor. Unpublished.

[14] Calder, P., and Linton, M. Managing Attributes in a User Interface Toolkit. Unpublished.

[15] Lalonde, W., and Pugh, J. Inside Smalltalk Volume II. Prentice Hall, pp 7-12, 1991.

[16] Palay, et al. "The Andrew Toolkit: An Overview". The Andrew Project: Selected Technical Papers. Information Technology Center, CMU, 1990.