

**A FRAMEWORK FOR THE RAPID PROTOTYPING OF  
ZOOMABLE USER INTERFACES**

Michael Bennett

A thesis submitted in partial fulfillment of the  
requirements for the degree of  
Masters of Science  
in the Department of Computer Science  
University College Dublin

## Abstract

Zoomable user interfaces (ZUI) are a superset and potential successor of the familiar Windows, Icons, Menus and Pointers (WIMP) desktop graphical user interface paradigm. However, the process of creating them remains a complex problem. The interplay between the information in the zooming space, the changing semantic content of the information based on scale and novel interaction techniques result in making designing and implementing ZUIs more challenging than for two dimensional interfaces. How can the process of creating ZUIs be simplified?

In this thesis the *Objects, Regions, Relations and Interface Logic (ORRIL)* framework is defined and presented as a technique for aiding ZUI design and creation. ORRIL makes explicit the data that appears in a zoomable information space, while simultaneously emphasising the relationships between user actions and transforms of the data. This is important for modelling and clarifying the processes that occur within a ZUI application. From the perspective of an implementer it may be used as an underlying model when designing and implementing a framework for building ZUIs.

The framework was generated by identifying the core requirements, components and functionality needed in a ZUI. Following this was a conversion of the results of the identification into a formal abstraction (ORRIL). This abstraction was then used as the basis for an experimental rapid prototyping tool (Nutmeg) for creating medium to high fidelity ZUIs. Nutmeg and indirectly ORRIL were evaluated to establish how successful they were at simplifying the process of creating ZUIs. Evaluation was carried out by using Nutmeg to implement a range of ZUI interaction techniques and implement a prototype audio-visual ZUI application (Media Dive) for browsing collections of songs.

## **Acknowledgements**

This thesis would not have occurred without the generosity of numerous people. I am especially indebted to my supervisor Dr. Fred Cummins who gave me the opportunity to pursue my passion.

Thanks also to Dr. Sile O'Modhrain for her support and enticing questions, and everyone in MIT's Media Lab Europe for interesting chats, constructive feedback and inspiring ideas. Not least of all everyone in the Adaptive Speech Interfaces Group and the Palpable Machines Group.

A big thanks to Rois (who I do believe may have gained knowledge of HCI by osmosis)!

Of course this work would not have been possible with the scholarship funding provided by the Higher Education Authority in Ireland. Thank you – it was much appreciated.

# Table of Contents

<b>ABSTRACT</b> .....	2
<b>ACKNOWLEDGEMENTS</b> .....	3
<b>TABLE OF CONTENTS</b> .....	4
<b>1. INTRODUCTION</b> .....	<b>9</b>
1.1. Introduction.....	9
1.2. Zoomable User Interfaces.....	9
1.3. Issues with Implementing Zoomable User Interfaces.....	10
1.4. Why Simplify ZUI Prototyping .....	12
1.5. Approach .....	12
<b>2. LITERATURE AND PRIOR WORK</b> .....	<b>15</b>
2.1. Introduction.....	15
2.2. User Interfaces .....	15
2.3. Key Zoomable User Interfaces .....	16
2.3.1. Spatial Data Management System .....	16
2.3.2. Pad.....	18
2.4. Developing Zoomable User Interfaces.....	20
2.4.1. Pad++.....	20
2.4.2. Jazz.....	20
2.4.3. Piccolo .....	21
2.5. Authoring Zoomable User Interfaces.....	22
2.5.1. MuSE.....	22
2.5.2. Space-Scale Diagrams .....	23
2.5.3. Tioga-2 and Counterpoint.....	25
2.5.4. Automating Interface Design.....	26
2.6. Conclusions .....	26
<b>3. ZOOMABLE USER INTERFACE REQUIREMENTS</b> .....	<b>28</b>
3.1. Introduction.....	28
3.2. Decomposing ZUIs .....	28

*Table of Contents*

---

3.3.	Requirements Defined.....	28
3.4.	Basis for the Requirements.....	29
3.4.1.	R1: Render .....	29
3.4.2.	R2: Place.....	30
3.4.3.	R3: Allow.....	30
3.4.4.	R4: Constrain .....	32
3.4.5.	R5: Position.....	33
3.4.6.	R6: Transform.....	34
3.4.7.	R7: Create .....	35
3.4.8.	R8: Define.....	36
3.4.9.	R9: Encode.....	38
3.4.10.	R10: Enable.....	38
3.5.	Examples of the Requirements in Media Dive .....	39
3.5.1.	Requirements in Media Dive .....	39
3.6.	Grouping the Requirements.....	41
3.6.1.	Why Simplify.....	41
3.6.2.	Further Analysis: Display, Interaction, Results, Data.....	42
3.6.3.	From Requirements to Requirement Groups.....	43
3.7.	Conclusions .....	44
<b>4.</b>	<b>OBJECTS, REGIONS, RELATIONS AND INTERFACE LOGIC .....</b>	<b>45</b>
4.1.	Introduction.....	45
4.2.	Why ORRIL.....	45
4.3.	A Usable Framework .....	45
4.4.	ORRIL Defined.....	46
4.5.	ORRIL Diagrams .....	47
4.6.	From Requirements to ORRIL .....	49
4.6.1.	Objects.....	50
4.6.2.	Regions .....	51
4.6.3.	Relations .....	51
4.6.4.	Interface Logic .....	51

*Table of Contents*

---

4.6.5.	Degree of Relationship Between the Components.....	52
4.7.	Examples of ORRIL.....	53
4.7.1.	Using Only Objects .....	53
4.7.2.	Activating a Song .....	53
4.7.3.	Semantic Zoom .....	55
4.8.	Conclusions .....	55
<b>5.</b>	<b>NUTMEG: A TEST IMPLEMENTATION OF ORRIL .....</b>	<b>56</b>
5.1.	Introduction.....	56
5.2.	A New Tool .....	56
5.3.	Overall Structure.....	57
5.4.	Capabilities and Functionality .....	57
5.4.1.	Basic ZUI Environment.....	58
5.4.2.	Applet and Application.....	58
5.4.3.	ZIML DOM .....	59
5.4.4.	Multiple Audio Formats and Streams .....	59
5.4.5.	Numerous Image Formats.....	59
5.4.6.	Scripting Languages .....	60
5.5.	Design and Implementation.....	61
5.6.	Conclusions .....	63
<b>6.</b>	<b>FROM ORRIL TO ZIML.....</b>	<b>64</b>
6.1.	Introduction.....	64
6.2.	Converting from ORRIL to ZIML .....	64
6.3.	Separation of Content, Layout and Logic .....	64
6.4.	Objects.....	65
6.5.	Regions .....	66
6.6.	Relations .....	67
6.7.	Interface Logic .....	68
6.8.	An Example of ZIML.....	69
6.9.	ZIML's XSD.....	72
6.10.	Conclusions.....	72

*Table of Contents*

---

<b>7.</b>	<b>PROTOTYPING APPLICATIONS WITH NUTMEG .....</b>	<b>73</b>
7.1.	Introduction.....	73
7.2.	Media Dive .....	73
7.2.1.	Functionality of Media Dive.....	73
7.2.2.	Using Media Dive .....	75
7.2.3.	Two Approaches to Constructing Media Dive .....	76
7.2.4.	Evaluation of the Approaches.....	79
7.3.	Implementing Different Forms of Zooming.....	81
7.4.	Conclusions .....	83
<b>8.</b>	<b>CONCLUSIONS AND CONTRIBUTIONS.....</b>	<b>84</b>
8.1.	Conclusions .....	84
8.2.	Contributions .....	86
8.3.	Future Directions .....	86
<b>A.</b>	<b>XML SCHEMA DEFINITION FOR ZIML .....</b>	<b>89</b>
<b>B.</b>	<b>DEFINITION OF TERMS .....</b>	<b>95</b>
	<b>BIBLIOGRAPHY .....</b>	<b>100</b>



# Chapter 1

## 1. Introduction

### 1.1. Introduction

Zoomable User Interfaces (ZUI) are a superset and potential successor of the familiar Windows, Icons, Menus and Pointers (WIMP) desktop graphical user interface paradigm (see Section 2.2). However, the process of creating them remains a complex problem. The interplay between the information in the zooming space, the changing semantic content of the information based on scale and novel interaction techniques result in making designing and implementing ZUIs more challenging than for two dimensional interfaces. How can the process of creating ZUIs be simplified?

### 1.2. Zoomable User Interfaces

In the late 1970s and early 1980s there was ongoing research into what were called multiscale interfaces [13] and which are now referred to as zoomable user interfaces [41].

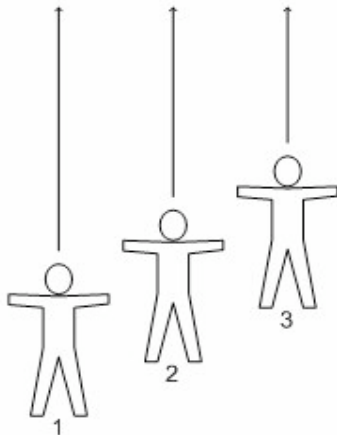


Figure 1-1. Spatially interacting with a bookcase.

ZUIs build on some of the spatial techniques [13, 41] we use to interact with the physical world around us. Imagine you are standing in a large room (Figure 1-1) and on the other side of the room is an unfamiliar collection of books in a bookcase. From a distance, at position 1 in Figure 1-1, you can discern that there are books in the bookcase, but it is not possible to read either the titles or authors. As you move towards the bookcase, to position 2, the apparent size of the books and bookcase increases, as well as the amount of detail that can be read. By this stage it might be possible to read one or two book titles, that is if the title font size is big and the contrast between the text and background colour is high. Once at position 3 it becomes possible to read the book titles and authors names. Also at position 3 the number of potential interactions with the books and bookcase increases. For example, if a book catches your eye you could pick it up

and read the book's abstract – at position 1 and position 2 it was not possible to physically manipulate the book.

Some common interactions that are fundamental to ZUIs occur in this example. Firstly, there is the variation in information representation as a function of scale – otherwise known as semantic zooming [41]. Secondly, there are the variations in possible interactions as a function of scale – this property does not have a clearly defined name, so from here on it will be referred to as “scalable interaction”.

Building on the example: imagine that the large room is replaced with a virtual three-dimensional room, and you are replaced with an avatar that you control. By moving the avatar's position your view of the room changes – you view the room through the avatar's eyes. At this point there is little difference between ZUIs and 3-Dimensional Virtual Reality. Next the movement of the avatar is restricted such that it can only be moved up and down, left and right, and backwards and forwards but cannot be rotated around any axis. The effect of these restrictions is that the room can only be viewed and explored in a constrained way. This limited way of perceiving and interacting with the virtual room is at the core of ZUIs.

Conceptually a zoomable user interface is an infinite three-dimensional space that users can interact with in a constrained manner. A user controls a viewport in the three-dimensional space. Movement of the viewport can only occur backwards and forwards along the Z-axis, left and right along the X-axis, and up and down along the Y-axis. Rotation of the viewport around the X-axis and Y-axis is not possible, and it is not normally possible around the Z-axis. Objects, such as images (or software applications), can be placed at varying locations within the three-dimensional space. A user can explore these images by moving the viewport around, e.g. move the viewport along the Z-axis and an image gets smaller or bigger.

### **1.3. Issues with Implementing Zoomable User Interfaces**

Creating zooming user interfaces involves a diverse range of skills, knowledge and experience in vastly different domains. It is a complex task. For example, designing and implementing an interface can involve taking into consideration the end user's physical, perceptual and cognitive capabilities, their cultural backgrounds and levels of education, their prior experience and context of use for the interface. Not least of all is the time and technical skills necessary for building the interfaces. How can the process of ZUI creation and implementation be simplified and made easier? In particular how can the process of

implementing prototype ZUIs be simplified?

An important aspect of constructing ZUIs is designing for the display of information at user controllable levels of detail, i.e. at different scales. This method of displaying varying information based on scale in ZUIs is called semantic zooming. The choice of what information to display is often dependent on the position of the user's viewport along the Z axis within the zoomable information space (Figure 1-2).

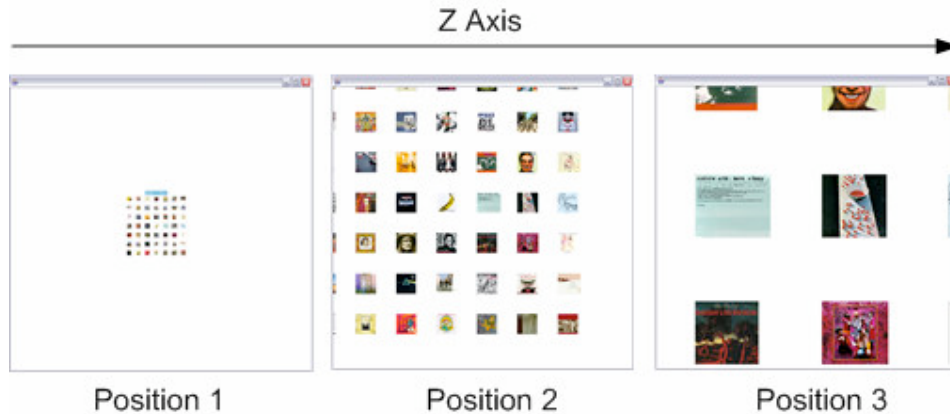


Figure 1-2. Viewport position along the Z axis affects the scale the information is displayed at.

The role of scale in ZUIs makes them more complex to create than WIMP interfaces. In WIMP interfaces all applications usually exist at the same scale and when zooming is used it occurs in a limited form, i.e. increase or decrease the size of the contents of a document or an image. The behaviour of a WIMP interface does not change with scale – that is there is no scalable interaction.

There is a considerable body of research into implementing WIMP interfaces [35, 36] that is relevant to ZUIs. This body of research touches on many questions including “What should WIMP interfaces be built from?” For example, should implementers of WIMP interfaces be able to build interfaces from pre-created widgets – such as buttons and windows? Or should WIMP interfaces be constructed from events, behaviours and pictorial representations that have to be composed into widgets [29], which in turn may be composed into interfaces?

The WIMP interface implementation questions also apply to ZUIs except there is a greater degree of complexity introduced by scaling and the associated semantic zooming and scalable interaction. For example, a ZUI could present what looks and behaves exactly like a WIMP interface at one level of scale yet when the user zooms in or out the interface could radically

changed style, behaviour and looks. How does an implementer cater for this? Should an implementer create many versions of an interface for each different scale (this presumes scaling is discreet rather than continuous)? Or does the implementer create a single interface and attempt to design an algorithm that automatically adapts the interface as a function of scale?

Consequently implementing ZUIs can be harder than implementing other types of interfaces. What can be done to simplify this process? More specifically what can be done to simplify the process of creating ZUIs without presuming particular ZUI types and behaviours? At this point in time it is important not to presume any particular ZUI type and behaviour because there are still many unanswered interaction and interface questions about ZUIs.

#### **1.4. Why Simplify ZUI Prototyping**

Digging a hole with a toothpick, while not an impossibility, would be a very hard task – made all the harder by not using the right tool. The idea of using the right tool for a job applies to researching ZUIs. Without the right tools an unnecessary amount of time and skills can be misdirected and wasted constructing ZUIs. Time which would be better spent quickly implementing and testing specific ZUI interaction techniques and displays.

A good tool need not only be something tangible. It may also be that which provides a conceptual framework for thinking about a problem – in this specific case thinking about constructing ZUIs.

The potential benefit from simplifying the process of creating and implementing ZUIs is that it could enable a greater number of researchers to examine ZUIs, which in turn could potentially encourage the wider adoption of ZUIs as various user issues and human-computer interaction questions are understood, dealt with and resolved.

#### **1.5. Approach**

The research and this thesis are structured as follows. A review of work related to creating and interacting with ZUIs and interfaces is presented. Then the core requirements, components and functionality needed in a ZUI are identified and defined.

This identification is done to establish and understand what the requirements are when creating and building ZUIs. Understanding this enables clarification of what is needed in ZUI construction. For example, a ZUI needs a virtual three-dimensional space that supports the

placement of images and audio sources.

Following this the identified requirements are analysed to create a simpler abstraction. The resulting abstraction is called ORRIL (Objects, Regions, Relations and Interface Logic) and serves as a framework for aiding ZUI design and creation. ORRIL makes explicit the data that appears in a zoomable information space, while simultaneously emphasizing the relationships between user actions and transforms of the data.

Then the question of “How successful and useful is ORRIL?” is examined by converting ORRIL from an abstract model into a test implementation. Creating a test implementation helped establish and evaluate the validity and expressiveness of ORRIL for defining and creating ZUIs. This test implementation took the form of an experimental tool, called Nutmeg, for rapidly prototyping medium to high fidelity zoomable user interfaces. Nutmeg helped establish what range and style of ZUIs could be implemented with ORRIL.

Nutmeg functions somewhat like a web browser. ORRIL was converted into a markup language, called the Zoomable Interface Markup Language (ZIML). Nutmeg reads in ZIML and parses and stores it in a Document-Object Model (DOM). This DOM is used to layout and render an interactive ZUI on display devices. The displays can be visual and aural. Interface and programming logic can simply be added to the ZUIs via a range of familiar scripting languages. Media that appears in a ZUI, referenced via the ZIML, can be images, audio and blocks of text. A diverse range of widely used image and audio formats are supported, and the images and audio can be stored on remote machines and accessed via standard network protocols.

How ZIML derives from and is based upon ORRIL is then scrutinized and explained. Implementing ORRIL as a usable framework means a close examination of each of the four ORRIL components had to occur. Exactly what are the properties of the components? Are four components enough for creating a usable markup language for describing ZUIs?

Finally a test ZUI and ZUI interaction techniques were implemented in Nutmeg and ZIML. This contributed to understanding the question: How usable is Nutmeg for creating ZUIs? By examining this question we indirectly evaluated the capabilities of ORRIL. In particular the differences in two approaches to implementing an application called Media Dive were highlighted and examined.

Thus in this thesis the question of simplifying ZUI creation is examined by:

1. analysing what is required for constructing ZUIs
2. creating an uncomplicated formalism based on what is learnt from the analysis
3. turning the formalism into a simple and usable framework
4. implementing the framework as a prototyping tool
5. attempting to use the prototyping tool to implement ZUIs and some common ZUI interaction techniques.

The core contribution of this work is a framework for rapidly prototyping ZUIs – this contribution takes the form of the Requirements, Requirement Groups, ORRIL, ZIML and Nutmeg.

# Chapter 2

## 2. Literature and Prior Work

### 2.1. Introduction

In this chapter the concept of a user interface is introduced and its development from the early days of computing to what most people are familiar with is briefly covered. The seminal implementations of Zoomable User Interfaces are outlined, which helps establish what kinds of ZUIs users may require. Then the process of creating ZUIs is examined from the perspective of important developer toolkits and libraries for implementing ZUIs. Authoring environments for enabling non-programmers to create ZUIs are discussed, as are some approaches for automating and guiding users in the process of creating ZUIs.

### 2.2. User Interfaces

Many of the computer based user interfaces we experience in our daily lives are predominantly based on research carried out in the 1960s and 1970s. Around 1963 Ivan Sutherland created Sketchpad [46] as part of his MIT PhD; this is widely recognised as the first GUI and featured zooming and the ability to sketch interfaces. In 1968 Douglas Engelbart led a group in Stanford that produced NLS [15]; this system featured concepts such as a mouse and multiple windows. Shortly thereafter Xerox PARC brought together a team of researchers who advanced the state of the art in numerous fields of computing, and produced innovations like overlapping windows, personal computing and What-You-See-Is-What-You-Get (WYSIWYG) editing [56].

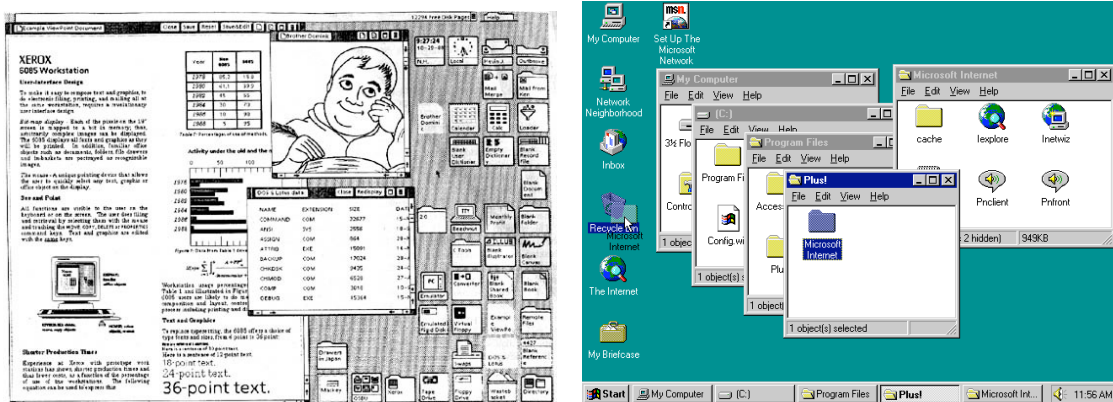


Figure 2-1. Left hand screen shot shows Star desktop, right hand screenshot shows Microsoft Windows.

The most familiar user interface legacy of that early work is the WIMP [48, 35] style interface. WIMP stands for Windows, Icons, Menus and Pointing device. Very little has changed in the underlying WIMP fundamentals in the many years since the WIMP interface was initially created and implemented in Xerox PARC.

One of the first iterations of the WIMP interface appeared as part of the Xerox PARC's Alto system [49, 35] and shortly thereafter was further developed as part of a commercial system called Star [45]. Windows were present in the Star GUI (Figure 2-1), as were icons, and menus. A prototype pointer, a mouse, was also available and could be used, in conjunction with a keyboard, to interact with the windows, icons and menus. Aesthetically the interface was ugly, but this was one of many limitations imposed by the technology at the time. Computers and the displays were not yet powerful enough to render visually rich graphics in real-time.

Apple's Lisa was the first widely sold computer that shipped with the WIMP style graphical user interface (GUI). To this day it is still the type of interface in use as the Apple and Microsoft GUIs (Figure 2-1).

A large amount of research has been carried out on optimization problems in WIMP GUIs, e.g. how to layout graphics and text to improve target acquisition [32], etc. Yet, choosing to use a WIMP interface is like deciding to wear a full-body straight jacket, block your sense of smell, remove your sense of taste and generally limit and ignore, in an extreme way, human physical and cognitive capabilities. It can easily be argued that current interfaces in wide use fall far short of the original vision of "user interfaces" put forward by Vannevar Bush in the 1940s [11].

### **2.3. Key Zoomable User Interfaces**

#### **2.3.1. Spatial Data Management System**

One of the earliest, and maybe the first, research project into ZUIs was the Spatial Data Management System (SDMS) [13], which was carried out for a number of years following 1976. Nicholas Negroponte and Richard Bolt in the Architecture Machine Group at MIT lead the research [9]. At that early stage the term zoomable user interface was not in use, instead this type of interface was referred to as a multiscale interface.

Their aim was to build on the spatial perceptual and organizing abilities of humans by

“concentrating on those qualities of interactive systems which may be a priori understood by a user. By this we mean that we will draw upon those experiences and concepts already familiar to the user. A data base management system, the Spatial Data Management System (SDMS), has been developed to address many of the above notions of spatiality. In this system we have enlisted the user’s a priori understanding of space for purposes of managing a very large data base, accessed in a comfortable and natural manner, and drawing upon the same perceptual abilities used by humans to remember and to navigate about the real world. These include the visual, aural, spatial, and haptic perceptions of the user.” [13]

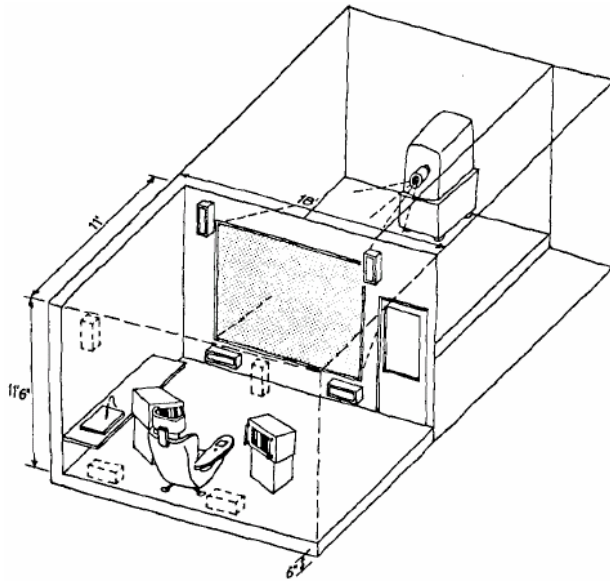


Figure 2-2. Design and layout of SDMS Media Room.

Over multiple iterations they conceived of and constructed a Media Room (Figure 2-2) that was used to interact with a zoomable information space. In this room a user could sit in a control chair in front of three computer screens. Two of the screens had a display size of 12 inches diagonally and were placed to the left and right of a large six foot by eight foot screen. The small screen on the right functioned as a navigation aid – it showed a world map that gave the user an overview of where they were in the zoomable information space. Shown on the large screen was a small part of the zoomable information space, i.e. what the user was focused on. The screen on the left gave the user information about the data they were interacting with. Direct manipulation, often with physical gestures, of the contents displayed on the touch sensitive small screens was sometimes available, usually depending on the type of content displayed on the large screen. Movement within the zoomable information space was done with two joysticks embedded in the arms of the control chair. Moving the left joystick controlled zooming, and moving the right joystick controlled x and y motion. Rates of movement were directly proportional to the pressure applied to the joysticks. By touching the screen displaying the world map a user could jump from location to location, this was referred to as “rapid transit”. Audio was used in the zoomable information space as a data

type and as a navigation aid. Eight spatially arranged speakers played the audio and a microphone captured audio. Written annotations, through a touch sensitive pad, and audio annotations could be added to locations in the zoomable information space. Photographs, diagrams, slides, text, movies, animated sequences, sound, and annotations could be placed in the zoomable information space.

SDMS is an impressive piece of work. Within it were most of the properties that were later on formally identified by other groups as traits of ZUIs. A number of criticisms can be levelled at SDMS.

SDMS was used to display and interact with mostly static data – it did not enable the user to use spatially located interactive software applications in the zoomable information space. This is mentioned in future work as “process as data” but considering the importance of software applications for interfaces this would seem to be too important an aspect of interfaces to have left unexamined.

The second criticism is that zooming was confined to geometric zooming, that is semantic zooming, which would seem to be very useful in ZUIs, was not examined. There are a number of possible reasons for this, ranging from the large amount of computational resources required for semantic zooming, to the need for (still under-developed) techniques and algorithms for automatically generating scale appropriate semantic representations of many different data types.

Thirdly, anecdotal evidence is presented to justify the benefit of SDMS but where is the formal evaluation of this? This criticism presumes a formal evaluation would have benefits.

Finally, and this is an indirect but important criticism, SDMS demonstrated a potentially better alternative to the WIMP interface yet why were ZUIs destined to be reinvented multiple times over the coming decades? Did the fact that SDMS required four networked computers, multiple screens and custom developed hardware mean that it was simply too computational expensive to be considered for mainstream users? Could it have been made work with fewer resources, and if this had been the case would our experiences with desktop interfaces now be very different – at the very least taking advantage of our spatial abilities?

### **2.3.2. Pad**

During the early 1990’s Ken Perlin and David Fox, from New York University, put forward

and developed what they stated is “*a new computer interface model called Pad*” [41]. At its core this work was a zoomable user interface with some extensions.

There are a number of reasons why this work was significant. It led to a rebirth of research into and increased awareness of ZUIs. Also the properties of ZUIs were more formally identified and defined. Two interface and interaction techniques that were put forward are semantic zooming and filter portals.

Semantic zooming is the process of displaying different representations of the same data as a function of scale. For example, when very far away from an electronic document a person would see the document’s title, as the person zooms closer an increasingly elaborate synopsis of the document is displayed, till eventually the person is viewing the complete document.

Portals are views into regions of the zoomable information space, and may be controlled by a user. Portals may be stacked and more than one portal can be displayed at once. Filter portals are portals that transform the representation of the content they are displaying. An example of this is a filter portal placed over a table of numbers could analyze the numbers and show the user a pie chart instead of the table.

Filter portals and semantic zooming break away from merely replicating a real world space as the basis for a ZUI. Instead they enhanced spatial interaction in ZUIs in a way that does not have direct parallels in the real/physical world.

To create a ZUI with Pad users could specify where Pad Objects are positioned in a zoomable information space (Pad Surface). A Pad Object was defined as “*any entity that the user can interact with (examples are: a text file that can be viewed or edited, a clock program, a personal calendar)*” [41]. Pad Objects were not only entities that could appear in the interface, they could also be filter portals. Therefore multiple Pad Objects could be composed in various ways to create user interface experiences.

The main criticism of Pad is that questions about how users interact with complex applications in a ZUI seem to be unexamined, e.g. if an application in the zoomable information space is undergoing semantic zooming does this impact with how the user can interact with the application? Does semantic zooming apply to interaction as well as representation? The same question applies to portals and filter portals.

Another criticism is that Pad was mono-modal. There does not seem to have been any

consideration given to the role of audio, or gestures, etc.

A secondary criticism of Pad has to do with the claim that the work is a new computer interface model; this seems too strong a claim. What would have been more reasonable is the claim that they have developed a variation and refinement of ZUIs. Though saying this they do reference SDMS in prior work.

## 2.4. Developing Zoomable User Interfaces

There is a considerable body of work on tools and frameworks [37], User Interface Management Systems (UIMS), UI toolkits and interface builders, for creating user interfaces, e.g. InterViews [31], Garnet [38], Amulet [39], UIDE [16], HUMANOID [47] and many more. In this literature review I am only focusing on approaches to programming and authoring ZUIs, because they are most relevant to this work.

### 2.4.1. Pad++

Pad++ [7] was a continuation of the work done on Pad. In Pad++, when compared to Pad, there seems to have been a greater emphasis on asking how the process of ZUI construction can be improved: *“Pad++ is a general-purpose substrate for exploring visualizations of graphical data with a zooming interface. While Pad++ is not an application itself, it directly supports creation and manipulation of multiscale graphical objects, and navigation through the object space.”* Pad++ was developed in mixture of C++ and Tcl/Tk [40], with scene rendering taken care of by the C++ code.

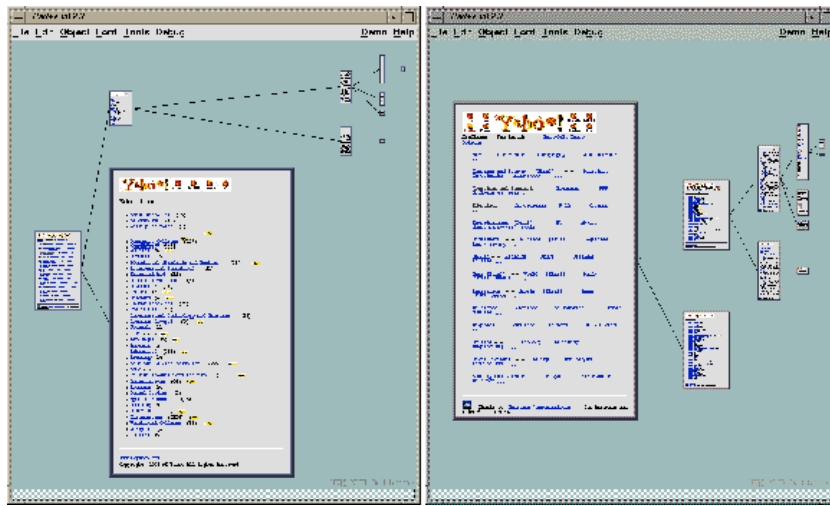


Figure 2-3. Screenshots of web browser implemented with Pad++.

With the Tcl scripting language a user of Pad++ could specify the layout and behaviours of zoomable interfaces. An issue with this is that there was no separation between defining interface layout and defining interface behaviours. This is an issue because it means more complexity in the resulting Tcl code (see [7], p22, “proc makeCalendar”). Though there may be a benefit to mixing layout and behaviour code, i.e. interface components could be coded to alter their layout in response to changes in scale (they called these components “*procedural objects*”).

Pad++ provided a range of objects to construct ZUIs with, these ranged from standard widgets, such as buttons to HTML renders (Figure 2-3).

### 2.4.2. Jazz

Jazz [6] built on the work in Pad and Pad++. “*Jazz is based on a ‘minilithic’ design philosophy. In Jazz, functionality is delivered not through class inheritance but rather by composing a number of simple objects within a scene graph hierarchy*” [6]. In later papers the Jazz design philosophy is referred to as polyolithic rather than minilithic.

With Jazz Bederson et al attempted to simplify the complexity of creating ZUIs by creating a Java library consisting of “*small, easily understood and reusable components*”. Jazz consists of a hierarchical structure of multiple classes, where the number and variety of classes and methods was less than was found in other widely used user interface toolkits, e.g. Java Swing [25], Microsoft MFC [33]. Jazz consisted of many small classes that could be composed in a scene graph hierarchy to create user interfaces.

When creating interfaces with Jazz it was noted that “*managing the polyolithic node structure was a significant programming burden*” [5] because developers had problems keeping track of the many small classes as they were added and removed from the scene graph.

The polyolithic scene graph approach may simplify the process of creating ZUIs, but it will primarily only do so for individuals familiar with Object Orientation and who have software development skills and experience.

### 2.4.3. Piccolo

Piccolo [5] is the most recent iteration in this series of ZUI toolkits. The design philosophy of Piccolo is monolithic. In Piccolo Bederson et al gave “*up on the idea of separating each feature into a different class, and instead put all the core functionality into the base object*

*class, PNode*". Their motivation for this decision was to reduce the amount of problems developers had when tracking multiple small classes in Jazz.

When contrasting (see [5], p11) Jazz and Piccolo it seems that Jazz offered more freedom for creating ZUIs but was more complex for developers, while Piccolo is more limited in the range of interfaces that can be created, but it is simpler for ZUIs programmers.

## 2.5. Authoring Zoomable User Interfaces

### 2.5.1. MuSE

The MultiScale Editor (MuSE) [20] differs significantly from the work described so far. Unlike the projects in Section 2.4, which were developer toolkits, MuSE is a domain independent authoring tool for creating ZUIs via direct manipulation. Furnas et al attempted to make scale an explicit dimension in MuSE – just like timelines make time an explicit dimension in video editing software. Early and less developed examples of applications for creating limited ZUIs through direct manipulation are Pad Draw [7] and HiNote [6].

Users of MuSE were presented with two main windows, the Space Editor (SE) and the Space Scale Editor (SSE). These windows were interdependent and depicted various properties of information in a zoomable information space. The Space Editor showed objects and information as it would appear to users in a zoomable information space. The Space Scale

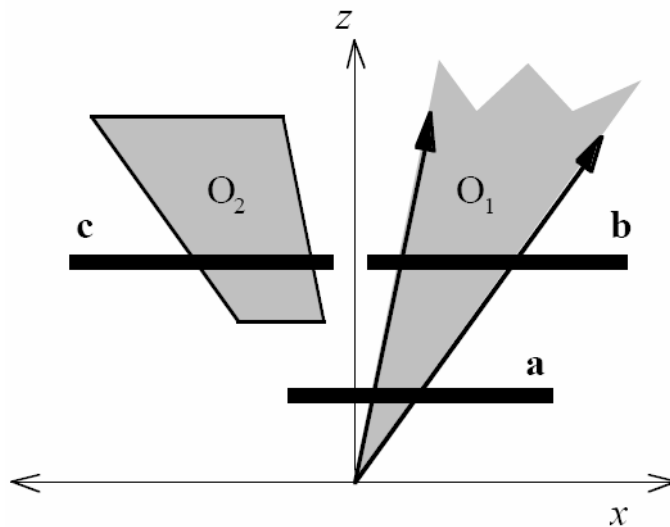


Figure 2-4. A 1+1D Space-Scale Diagram showing what kind of information would appear in MuSE's SSE window. A, b and c are viewports into the zoomable information space, while  $O_1$  and  $O_2$  are objects that appear in the zoomable information space.

Editor made scale more explicit by showing where information is laid out along the Z axis (Figure 2-4). In Figure 2-4, which comes from [20], you can see a top down view of the Z axis.  $O_1$  and  $O_2$  are objects that appear in the zoomable information space.  $O_2$  has a bounded box, which means it will only appear when the viewport (c) is within certain ranges along the Z axis. For more information on bounded box zooming see Section 7.3.

One of the more interesting features of MuSE was the ability to author pan and zoom trajectories by demonstration, and then edit the trajectories by direct manipulation. Pan and zoom trajectories are important in ZUIs because they can be used to create different kinds of interactions, i.e. by creating a zoom and pan trajectory over multiple objects a semantic zoom could be created.

A number of issues arose with MuSE, which would seem to be general problems with direct manipulation authoring tools for ZUIs. There was the problem of navigating and getting lost in the zoomable information space – this often occurs with ZUIs and is a known problem without a clear solution. Also there were issues with understanding the interdependencies between the SE and SSE windows. When the contents displayed in the SE window were changed or updated it was not always clear why the SSE window was correspondingly updated in a particular manner.

Furthermore there is no easy way of grouping zoomable objects and information. This is hard because objects in a group may exist at different scales. Therefore editing a group property that depends on scale could affect the individual objects in very different ways.

Other issues also arose; such as too many objects simultaneously displayed made it hard to understand what was selected for editing. Finally there was the issue with the inability to have a consistent snap-to-grid for objects. This is tricky to solve because the objects could all exist at different scales with different alignments, which means the grid size may or maybe not be dependent scale.

### **2.5.2. Space-Scale Diagrams**

Space-Scale Diagrams were used in MuSE to help make scale an explicit dimension. Space-Scale Diagrams are a diagrammatic formalism for understanding scale in ZUIs by “*representing both a spatial world and its different magnifications explicitly, the diagrams allow the direct visualization and analysis of important scale related issues for interfaces*” [19].

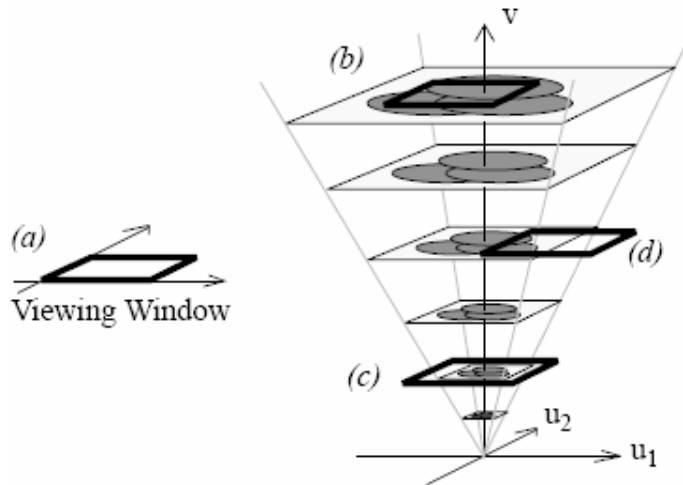


Figure 2-5. Multiple views of the same content at different scales are used to as the basis of Space-Scale Diagrams. C, d and b show what would appear in the viewing window “a” at different positions along the Z axis (v).

Space-Scale Diagrams can be thought of as helping clarify what would appear in a viewing window as the viewing window moves through a zoomable information space. If you look at Figure 2-5 (from [19]) you can see that at position “c” along the V vector (equivalent to the Z axis) the viewing window shows all the objects in the zoomable information space. When the viewing window is at position “d” there is only a partial view of the contents of the zoomable information space. When the viewing window is at position “b” there is a more limited view of the full contents of the zoomable information space, but the contents that do appear in the viewing window are larger.

Imagine if the viewing window “a” was treated as a single point and moved from the start of vector V to position “d”. This would trace out a trajectory, which I will refer to as  $T_1$ . A different trajectory, called  $T_2$ , would occur if the single point viewing window was moved between position “c” and position “b”. As the  $T_1$  and  $T_2$  trajectories are traversed different views of the zoomable information space are seen. Visually diagramming these trajectories and what appears along them is the basis for Space-Scale Diagrams. Figure 2-4 shows a 1+1D Space-Scale Diagram, for which there is an explanation of in the previous section and a more detailed explanation of in [20].

Space-Scale Diagrams offer a rich way of visually depicting scale in ZUIs, and their practical applications in MuSE has also been shown. A minor issue with Space-Scale Diagrams is how well they will work when there are a large number of objects in a zoomable information

space; this was discussed in [20].

### 2.5.3. Tioga-2 and Counterpoint

Tioga-2 [1] and Counterpoint [21] are examples of domain specific authoring tools for creating zoomable interfaces for particular tasks. Tioga-2's prime research focus was not the creation zoomable user interfaces but it did enable users to create a range of interfaces where zooming could be used to drill down into database datasets.

In particular Tioga-2 strove to make *“it much easier for database users to develop database applications”*. Users of Tioga-2 could construct the database applications with a data-flow box-and-arrows visual programming language [10]. With Tioga-2 users were able to create interfaces that could have semantic zooming and portals, though portals (see Section 2.3.2) were called *“wormholes”* and semantic zooming was referred to as *“drill down”*.

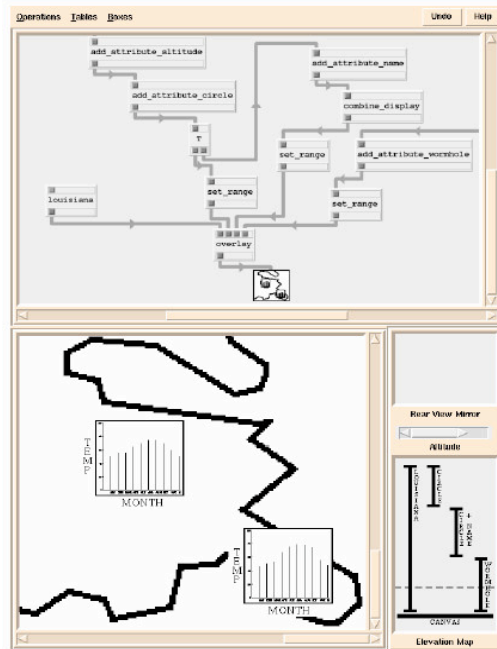


Figure 2-6. Screen shot of Tioga-2 where the top window shows the data-flow language and the bottom window shows a new interface.

Counterpoint was *“a tool to simplify the creation and delivery of zoomable presentations”*. With Counterpoint users could create zoomable slide show presentations, somewhat equivalent to Microsoft's Powerpoint [34] slide show presentation software. In fact Counterpoint was implemented as a Powerpoint plug-in

One of the key tasks in Counterpoint was laying out the presentation slides in the zoomable information space. A user could layout the slides with direct manipulation, but then there was the question of how the user can organise the slide traversal order. Creating the order the slides should be traversed was done by letting the user move around the zoomable information space and create a path through the zoomable information space. These paths resemble the trajectories discussed in Section 2.5.2. Then the user could layout the slides on the path and giving a presentation involved moving along the path.

The advantage of Tioga-2 and Counterpoint was they enabled people with little or no

programming skills to create zoomable user interfaces, though the types and ranges of the interfaces were limited. Tioga-2, with its visual programming language, may retain some of the flexibility of authoring ZUIs with a programming language while also having some of the advantages of a direct manipulation ZUI authoring tool.

### **2.5.4. Automating Interface Design**

Various research groups have asked: Can the process of creating ZUIs be automated? This has led to the development of an assortment of techniques. For example Woodruff et al in “Constant Information Density in Zoomable Interfaces” [53] outlined a method for automatically measuring the information density of a ZUI. This information density metric is then used to provide feedback to an implementer of a ZUI. The potential benefit of this feedback is that the implementer will create a better ZUI.

Another example is discussed in the paper “Nested Interface Components” [42]. In this paper Perlin et al present a way of grouping interface components into hierarchical structures that can be interacted with via zooming. These nested interface components are another way of creating semantic zooms with a form of visual programming. The hierarchical structures are equivalent to the trajectories and paths in MuSE and Counterpoint; except the nested interface components create a zoom of a particular set of components, rather than a zoom along a path in a zoomable information space. This implies a Nested Interface Component is partially independent of the zoomable information space in which it is located. It is a complete unit in itself, which could be useful for sharing reusable components among ZUIs.

## **2.6. Conclusions**

The end result of the work on Pad, Pad++, Jazz and Piccolo were predominately libraries and frameworks that should enable software developers to more easily create ZUIs. This is also the case with number of other ZUI construction research projects, such as Tabula Rasa [17], Zomit [43] and, indirectly, SATIN [22]. In nearly all these projects there was a strong emphasis on building ZUIs using some variation of Object Orientation (OO) and scene graphs. These toolkits may simplify and speed up the process of developing ZUIs but they are primarily only useful to people with software development skills and/or who have access to programming resources.

Some of the more recent work on toolkits for creating ZUIs, e.g. Ubit [29]; has moved towards a fusion of scene graphs with a declarative language, such as UIML [1] and XAML

[52]. This approach has the potential advantage of decouple programming logic from interface layout specifications, which could allow developers to focus on programming the logic behind interfaces and enable interface designers to focus on creating the user experiences.

MuSE, Tioga-2 and Counterpoint are prime examples of approaches that have been taken to simplifying the creation of ZUIs for non-programmers. These approaches tended to mix some form of direction manipulation of objects in a zoomable information space, coupled with methods for authoring relationships between and behaviours of the components. The disadvantage of these methods is that they can often be limited in the range and style of ZUIs that can be created with them. This is because complex control of the interface components can require custom programming.

Research to date on simplifying the creation of ZUIs can be broadly classified into low-level developer toolkits and high-level authoring environments, with some methods for automating and guiding the design of ZUIs.

Is it possible to strike a balance between the flexibility of the low-level approaches while retaining some of the simplicity of the high-level authoring environments for creating ZUIs?

# Chapter 3

## 3. Zoomable User Interface Requirements

### 3.1. Introduction

In this chapter the core requirements, components and functionality needed in a ZUI are identified.

Before an answer to “How can the process of creating ZUIs be simplified?” can be given we need to establish and understand what the requirements are when creating and building ZUIs. This understanding enables the clarification of what is needed in ZUI construction. Without an understanding of what is needed in the process of creating ZUIs there is no way of knowing how to simplify it.

### 3.2. Decomposing ZUIs

A ZUI can be decomposed into a set of separate and interdependent requirements. These requirements could be used as the basis for ZUI compositional units. In later chapters I use the identified requirement to extrapolate to a more general framework for ZUI creation. There are ten requirements that I have identified as needed for the creation of a feature rich ZUI. In the following sections I will examine where these requirements are derived from and the justifications for them.

### 3.3. Requirements Defined

The requirements are as follows:

- *R1: Render* a display that a user can perceive and interact with.
- *R2: Place* on the display at least one viewport into a very large three-dimensional information space.
- *R3: Allow* movement of the viewport in the information space so that it can pan and zoom.
- *R4: Constrain* the viewport such that it is impossible to rotate it.
- *R5: Position* data, such as text, images and audio, at specific spatial locations within

the information space.

- *R6: Transform* the data based on the viewport position and other occurrences, e.g. user actions.
- *R7: Create* a mapping between which occurrences trigger which transforms of the data.
- *R8: Define* areas within the information space where the mappings exist, i.e. not all transforms will be relevant to all locations or data.
- *R9: Encode* the nature of the transforms that occur on data.
- *R10: Enable* transforms and mapping to be altered.

It should be noted that not all applications need to fulfil every requirement, but all applications have to conform to more than half the requirements. This is elaborated upon later (see Section 3.5.1); sufficient to say for now R1 to R5 must be fulfilled to create even a simple ZUI. R1 to R10 can be thought of as a sliding scale, where increasing the scale enables increasingly more complex ZUIs to be built (Figure 3-1).

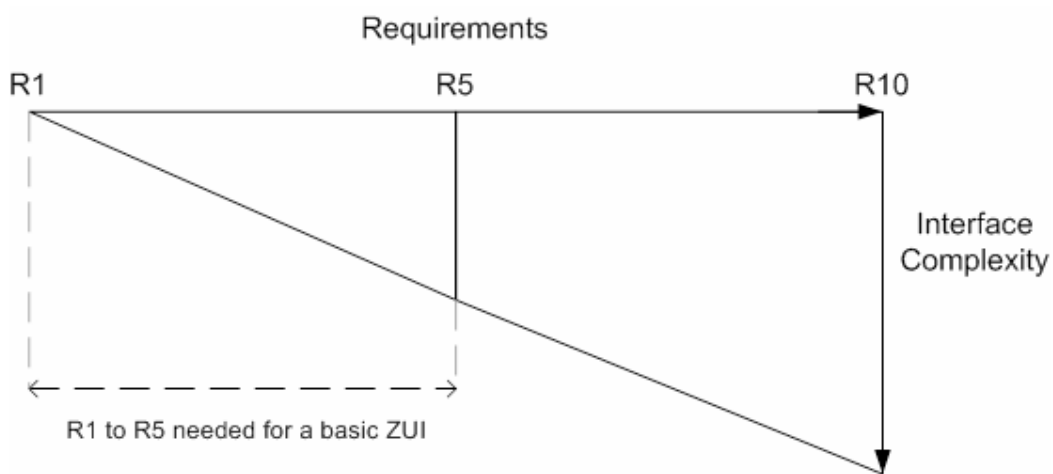


Figure 3-1. Relationship between fulfilling the Requirements and the complexity of the resulting ZUI.

## 3.4. Basis for the Requirements

### 3.4.1. R1: Render

Definition – *R1: Render* a display that a user can perceive and interact with.

The first requirement R1 is critical, if its not fulfilled then all the other requirements are effectively useless. Without a display there are no output channels that a user can use to perceive a ZUI.

The word “display” is used in a broad sense, in that it refers to more than visual displays – it may also refer to auditory displays, and potentially to other forms of displays, such as olfactory, haptic, etc.

In this work I have primarily focused on the creation of visual and auditory displayed ZUIs, while taking into consideration other display types. When terms such as “appear in the display” are used they should be considered in light of this fact.

#### **3.4.2. R2: Place**

Definition – *R2: Place* on the display at least one viewport into a very large three-dimensional information space.

R2 can be broken down into two interdependent sub-requirements; these are a) a viewport and b) a very large three-dimensional information space. Looking at the “b” sub-requirement first, we note that ZUIs depend upon three-dimensional information spaces, i.e. interface components in ZUIs are spatially organized along the X, Y and Z planes (data has a location). Very large information spaces must be supported because ZUIs may have to present interfaces consisting of millions of spatially arranged items.

Turning to the “a” sub-requirement we note that the three-dimensional information spaces can be very large, in fact they could be larger than any physical space. An effect of the large information space is users won't be able to perceive all the data in the space simultaneously as they could easily be overwhelmed. Methods for controlling how much content a user is exposed to are needed. Viewports serve as part of the solution, i.e. they can delimit and control how much of the information space the user is presented with at any one time.

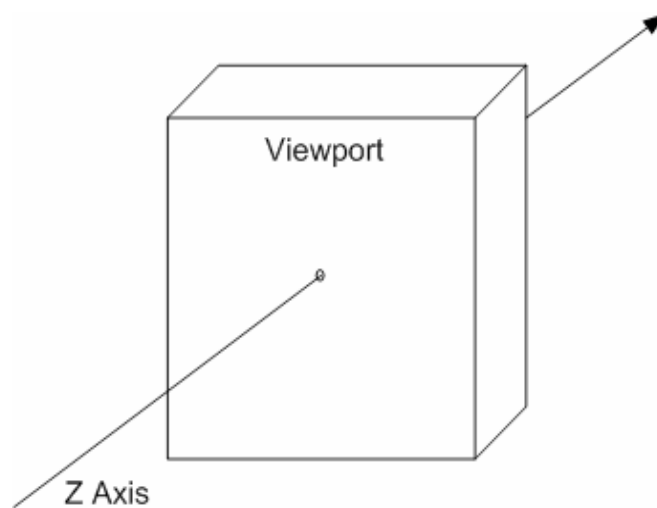
#### **3.4.3. R3: Allow**

Definition – *R3: Allow* movement of the viewport in the information space so that it can pan and zoom.

A viewport alone is not a sufficiently complete solution for controlling exposure to large collections of spatially arranged components. Control over what part of the large information

space is displayed within the viewport is needed. Otherwise a viewport provides only a fixed view of the content at a specific set of locations. Selective control of what content is displayed in the viewport can in many ways be thought of as enabling users to perform interactive “visual/auditory” queries of the content.

The methods and means of controlling the viewport help shape the user experience of interacting with the large information space – consider how panning a viewport is a familiar part of WIMP metaphor, i.e. we scroll a page of text up and down, and scroll a two-dimensional array of numbers in a spreadsheet left/right and up/down, etc. One of key differences between ZUIs and the WIMP metaphor is the ability to zoom. Zooming means moving the viewport backwards and forwards along the Z-axis (Figure 3-2), which alters the scale of objects that appear within the viewport. There are more complex variations of zooming, i.e. semantic, geometric, bounded, which will be touched upon later.



*Figure 3-2. Movement of the viewport along the Z axis.*

Updating the display of the viewport can be thought of in two ways. Firstly, the viewport is moving through a set of X, Y, Z locations and simultaneously displaying content at those locations, and, secondly, the content itself is changing location according to the viewport’s frame of reference. Both these perspectives would effectively create the same end user experience, but for low-level implementers of ZUIs changing the viewport position, rather than updating the location of thousands of interface components, is preferred because it would be less computational demanding. In reality the “movement” of the viewport should be a

fusion of the two approaches, with data items having a fixed location but only getting rendered and managed on a display when a viewport is showing the item’s locations.

**3.4.4. R4: Constrain**

Definition – *R4: Constrain* the viewport such that it is impossible to rotate it.

ZUIs and virtual reality share a common trait in that both use virtual three-dimensional spaces that are often very large. A problem with virtual reality is spatial disorientation [12], where users can easily become lost in the information spaces. There are numerous possible reasons for this, such as artificial physical metaphors, e.g. cityscapes, that break down when interacted with, the constantly changing nature of the information in the space makes location/reference based wayfinding harder [30], etc. In a sense a user has too much freedom to explore without the familiar physical constraints and clues imposed by our bodies and the environments we exist in.

Within ZUIs there is a deliberate constraint on how the large information spaces can be explored; specifically, there is a limitation on how the viewport can move around. This constraint reduces the degrees of freedom (DOF) of the viewport (Table 3-1). The constraint is such that the viewport cannot be rotated within the large information space; it can only be panned and zoomed on fixed axis.

*Table 3-1. Degrees of freedom of different styles of viewports.*

<i>Viewport Type</i>	<i>Move X</i>	<i>Move Y</i>	<i>Move Z</i>	<i>Rotate X</i>	<i>Rotate Y</i>	<i>Rotate Z</i>
Rotation Restricted	yes	yes	yes	no	no	no
Rotation Capable	yes	yes	yes	yes	yes	yes

An inability to rotate the viewport reduces the complexity associated with repeatedly moving between the same locations. Complexity is reduced because the variety of directions and orientations a component can be approached from is greatly reduced (from 6 DOF to 3 DOF). For example (Figure 3-3), moving a rotation restricted viewport from point A(5, 7, 2) to point B(14, 15, 9) need only be done with at most three DOF (left/right, up/down, backwards/forwards), while a rotation capable viewport increases this to potentially six DOF

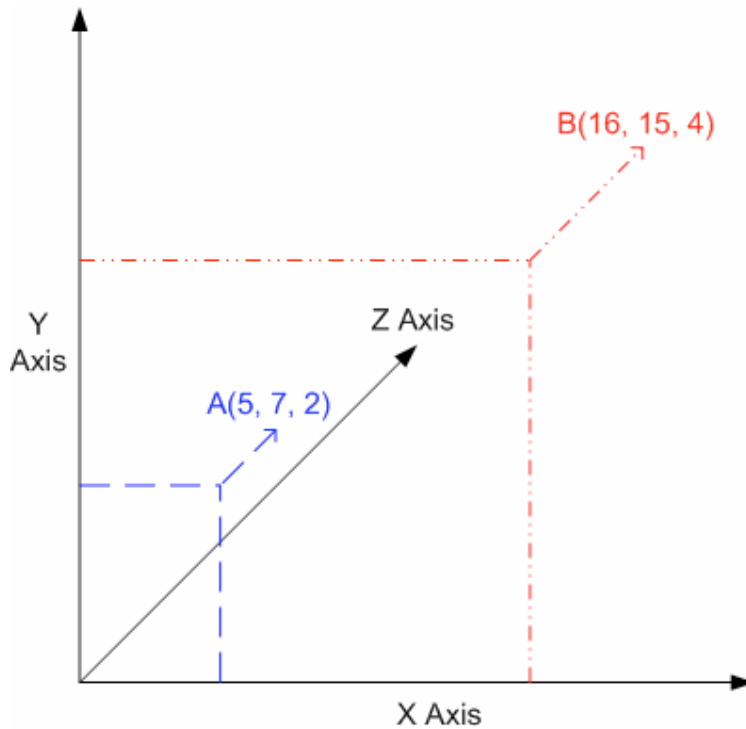


Figure 3-3. Number of potential paths between point A and point B vary on viewport DOF.

(left/right, up/down, backwards/forwards, rotate viewport around X axis, rotate viewport around Y axis, rotate viewport around Z axis).

### 3.4.5. R5: Position

Definition – R5: Position data, such as text, images and audio, at specific spatial locations within the information space.

Without any interface components or content a ZUI would serve no purpose. Therefore the ability to position content at specific locations within the large information space is a basic prerequisite.

How the content is positioned at specific locations varies on the content type, i.e. what anchor point should be used for attaching content to specific locations? For example, an image could be positioned based on using the upper left hand corner of the image as an anchor point. Blocks of text could be surrounded by a bounding box and the upper left hand corner of the bounding box used as an anchor, and audio sources may be treated as point sources. Consistently using the same style of anchor points with each content type is important. Otherwise a person placing content based at an X, Y, Z location may be unsure of the exact

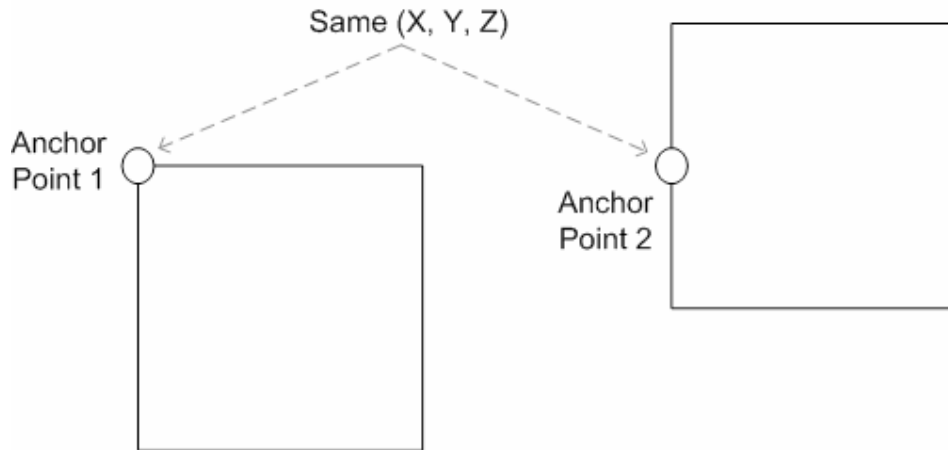


Figure 3-4. Same X, Y, Z but the square appears in different positions because different anchor points are used.

alignment between the content and the coordinate (Figure 3-4). The orientation/rotation of the content is usually aligned so that it appears in the viewport as though it is on a flat surface, though this does not need to be the case.

### 3.4.6. R6: Transform

Definition – R6: Transform the data based on the viewport position and other occurrences, e.g. user actions.

Transforming and updating the content displayed in user interfaces is a basic requirement; otherwise interfaces would be static and unresponsive to user actions and external occurrences. For example, updating the display of a digital clock on a WIMP desktop is a

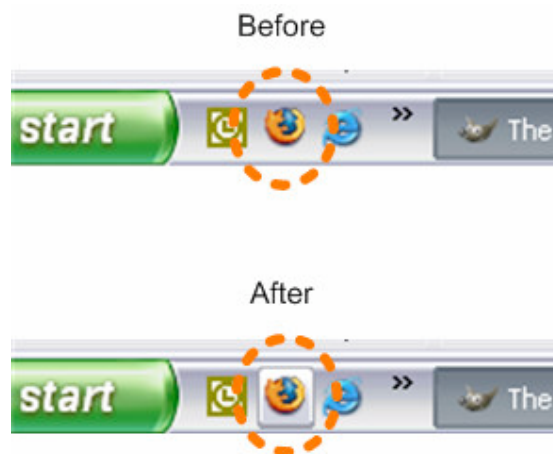


Figure 3-5. An example of a simple transform where an icon is highlighted when a mouse pointer passes over the icon.

basic transform that is trigger by an external occurrence, i.e. the hardware clock changing.

The range of transforms can vary from very simple to very complex. An example (Figure 3-5) of a simple transform is highlighting an icon that is displayed on-screen when the mouse pointer passes over the icon. A more complex interface transform is a menu system, where multiple sub-menus may be displayed in response to a user's actions.

Occurrences are defined as user actions, hardware triggers and interface events. Thus occurrences often signify a change of state that should be conveyed via the interface, i.e. occurrences are often responsible for starting and altering transforms. Occurrences and transforms form an open loop, i.e. an occurrence leads to a transform, which leads to an occurrence, etc. Part of the loop includes the users of the system (Figure 3-6).

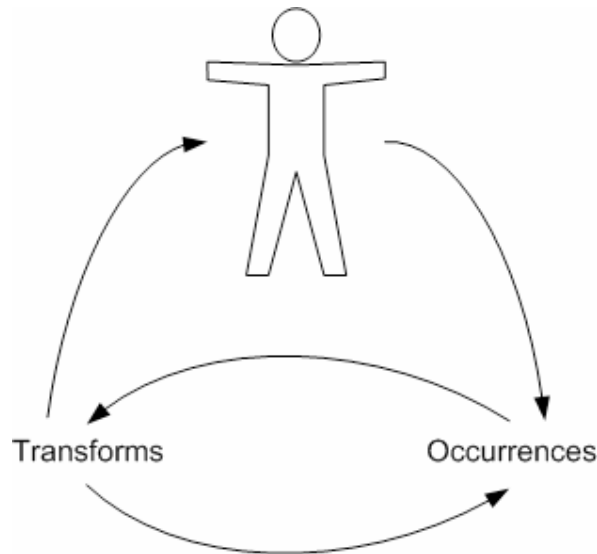


Figure 3-6. Transforms and Occurrences form an open loop with the user.

The set of possible occurrences varies as a function of the complexity of the interface, the range of interface events, the physical environment the interface and application runs on, and the range of hardware triggers. This relationship between occurrences and transforms is a key aspect in interfaces, without which interfaces would be static.

### 3.4.7. R7: Create

Definition – R7: Create a mapping between which occurrences trigger which transforms of the data.

As was just noted transforms and occurrences form an open ended loop where there can be multiple dependencies between transforms, occurrences and the users. Occurrences and transforms resemble the familiar cause and effect cycle we experience in the world around us, e.g. hold a tennis ball five feet above the ground, let go of the ball and it will drop to the ground.

In an interface this cause and effect cycle has to be explicitly created. There is no “natural” relationship between clicking on an image and having another image appear, which is how one can think of using a hierarchically structured menu system. Therefore part of designing and implementing an interface involves explicitly creating various cause and effect cycles. Some of these cause and effect cycles have become standard and widely used, e.g. buttons, menus and menu bars, scroll bars, etc.

An occurrence can be thought of as a cause, and a transform as an effect and creating interfaces involves defining mappings between occurrences and transforms.

#### **3.4.8. R8: Define**

Definition – *R8: Define* areas within the information space where the mappings exist, i.e. not all transforms will be relevant to all locations or data.

Occurrences often need to be associated with varying transforms depending on when and where they occur. For example, in a standard WIMP interface if a menu, that is already in a selected state, is re-selected then the result of this action may not be the same as selecting an unselected menu. In this case the interface behaviour is dependent on prior states. The cause and effect cycle can be conditional.

Within ZUIs interface behaviour can be a function of scale (see Section 1.2 “scalable interaction”). That is at one level of zoom (Figure 3-7) the interface has a certain set of behaviours (Interface Behaviour Set A – IBS A), while at a greater level of zoom the interface has a richer set of behaviours (IBS B), and at a still greater level of zoom the set of interface behaviours steadily increases (IBS C).

This highlights the fact that there needs to be a way of creating variable N:M mappings between occurrences and transforms to create interface behaviours. Though it is not as simple as that – for there needs to be a way of binding a particular instance of an occurrence to a transform, e.g. a single mouse click does not simultaneously apply to every button displayed

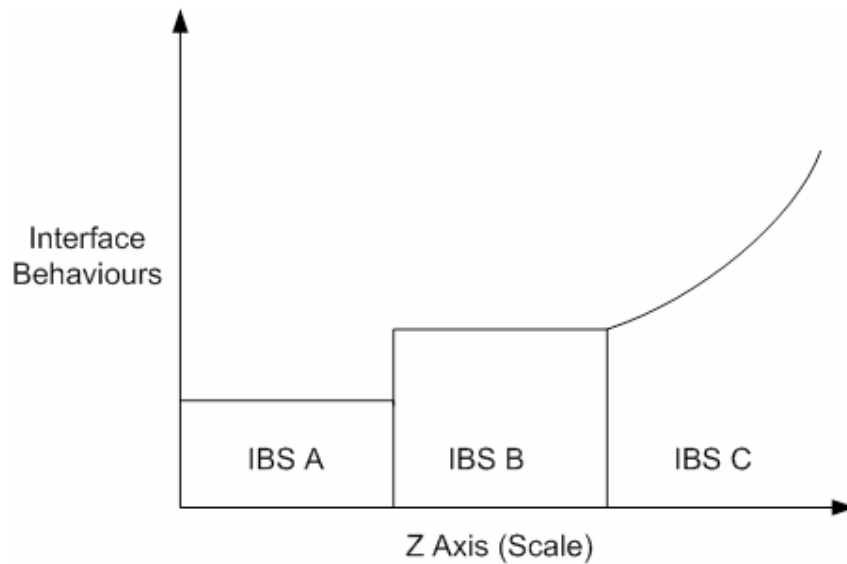


Figure 3-7. Relationship between scale and increasingly complex interface behaviours.

in a WIMP interface; the mouse click “acts” upon the button that was clicked.

This can be done in ZUIs by placing spatial constraints on the N:M Occurrence-Transform mapping. An occurrence may or may not be captured depending on whether it occurred within, or not within, a specific spatial range, i.e.

```
IF mouse button WAS clicked
  IF click OCCURRED
    BETWEEN x AND x + 10
  AND
    BETWEEN y AND y + 10
  AND
    BETWEEN z AND z + 10
  THEN
    trigger associated transform
  END IF
END IF
```

Therefore the ability to define regions that act as constraints on N:M Occurrence-Transform mappings is required. Possibly a larger number of dimensions (rather than only three) are needed to also encode and constrain aspects of prior and future states.

#### **3.4.9. R9: Encode**

Definition – *R9: Encode* the nature of the transforms that occur on data.

The preceding requirements analysis justified the role of a transform but did not help us understand how a transform should be encoded. If a transform is not encoded in some form then the triggering of a transform by an occurrence will have no effect. That is a transform would consist of no actions.

When encoding transforms the following properties and requirements need to be taken into consideration, a) transforms act upon objects, such as images, audio files, etc, b) a transform consists of an action, or multiple actions, and c) a transform needs to be storable, retrievable and executable, and ideally self-reflective, i.e. it can understand its own current state.

The structures affected by a transform are very dependent on the underlying model that is used to store information about the ZUI and related data (see Section 3.4.5). For example, does the underlying model record information on each pixel that appears in the ZUI, and if so does it provide access to each pixel or only provide limited forms of access to blocks of pixels, i.e. complete images?

This in turn impacts upon the granularity of the actions available for defining transforms. In the prior example a single atomic action could be to move a pixel, or move a group of pixels. Part of the process of deciding what actions to make available involves establishing how much control an implementer of a transform should have over the interface (pixel level versus image level control), and shaping how an implementer can go about building the interface.

#### **3.4.10. R10: Enable**

Definition – *R10: Enable* transforms and mapping to be altered.

In Section 3.4.9, it was stated that a transform needs to be self-reflective – this means that a transform should be able to query its current state, other occurrence's state and relationships with/between occurrences. The ability to query a state is required because there are interaction techniques that alter the behaviour of interfaces. If it were impossible to know the current

state of the system then it would be hard to know how to alter existing components of an interface in a predictable manner.

An example of one of these interaction techniques is the ability to drag an icon from one area of a screen to another area of a screen, while still associating the icon with the same actions, e.g. clicking on a Java icon starts Java no matter where the icon is on a WIMP desktop. In this example the position of the icon is altered and the relationship between actions and the icon is maintained. This is done by also moving the region that captures icon events, i.e. the icon has changed spatial position so clicking upon the icon occurs in a different spatial range (see Section 3.4.8).

Note: From this point onwards the ten identified requirements are referred to as the Requirements with a capital R.

#### **3.5. Examples of the Requirements in Media Dive**

Media Dive is a prototype audio-visual ZUI application that, as part of this work, was designed and implemented for browsing large audio collections, such as songs. Section 7.2 contains a detailed description of the functionality of Media Dive with screen shots. It would be advantageous to glance at that section before continuing.

An aspect of this research involved implementing Media Dive in two very different ways. Two different approaches were taken to serve as a contrastive analysis of “methods for constructing ZUIs”. The first version of Media Dive was built using Piccolo [5], a low level Java library that provides a rich and complex API for implementing ZUIs. The second version was built with Nutmeg (see Section 5), a framework and tool for rapidly prototyping audio-visual ZUIs. Nutmeg was created as part of the process of examining the feasibility and expressiveness of the Requirements and ORRIL (see Section 4) for defining and implementing ZUIs.

##### **3.5.1. Requirements in Media Dive**

Looking at Media Dive from the perspective of the Requirements occurring in Media Dive we note that there is a display, both audio and visual. This display means that Requirement R1: Render is met.

Within the display is a window (viewport) into a large information space where the images and audio can appear. The viewport and large information space jointly fulfil Requirement

R2: Place.

The viewport into the large information space can be panned and zoomed in a limited manner, thereby meeting the R3: Allow and R4: Constrain Requirements. Or the equal but alternative perspective could be taken that the images and audio sources can be panned and zoomed in the viewport, which also meets the R3: Allow and R4: Constrain Requirements.

Images and audio sources are spatially arranged in a grid in Media Dive. This spatial arrangement is along the X and Y axis, and along the Z axis. This means that the Requirement R5: Position is fulfilled.

By fulfilling Requirements R1 to R5 we have a basic ZUI, i.e. there is a display with a viewport that can display spatially arranged images and audio sources, which can be panned and zoomed.

The user can zoom and pan images and audio. Zooming towards the images results in a basic transform, i.e. the increase and decrease in the size of an image. Zooming in or away can alter the volume of the audio, and controls whether a piece of audio plays or not. The alteration of scale and volume means that R6: Transform is satisfied, i.e. an external Occurrence caused a Transform (see Section 3.4.6).

In Media Dive each piece of audio is associated with a specific image. This has the effect that when a zoom or pan occurs on an image there is a direct relationship between what happens to the audio and what image is displayed. The fact that this explicit relationship exists means there is compliance with the R7: Create Requirement.

Tied into the point in the previous paragraph is the Requirement R8: Define. As was pointed out a relationship exists between an image and an audio source, but this relationship is constrained. In that it only exists between certain ranges on the Z axis. For example, the volume of audio is not a concern till the audio source is playing, and the audio source only plays when an image is zoomed in above a specific scale. This range limitation uses the R8: Define Requirement.

Within Media Dive the nature of the transforms is simple. There is a transform controlling the stopping and starting of the audio, and another increasing and decreasing the volume of the audio. Stopping and starting is a discreet transform, while the increase and decrease is a continuous transform. These transforms are encoded in the form of computer code, thus

meeting the R9: Encode Requirement.

The R10: Enable Requirement is not currently used in Media Dive. However Media Dive could be extended so that a user could reorganize the layout of the images and audio sources by direct manipulation. This would mean the mapping between zooming in to a particular area in the information space and what audio sources are played would need to be altered in response to the user’s actions. Thus meeting the R10: Enable Requirement.

### 3.6. Grouping the Requirements

At this point we have the Requirements, which are fairly abstract and very interdependent. The next question is can they be simplified further? The answer is yes. The following sections show how this is done.

#### 3.6.1. Why Simplify

Stepping back and re-analyzing ZUIs in the context of the Requirements leads to the realization that an even more simplified framework can be developed. Furthermore the resulting simplification is at a higher level of abstraction which, in this case, means it is applicable to more general classes of interfaces, e.g. WIMP, audio and haptic interfaces, etc.

This simplification is done by carrying out a high level analysis of interfaces and relating this analysis to the Requirements – with the aim of classifying the Requirements into broader categories. A broader classification is beneficial because it means there are a smaller number of requirements, and a reduction in the number of relationships between requirements.

The end result of this simplification is the classification of the ten Requirements into Groups, as show in Table 3-2.

Table 3-2. How the Requirements are classified into the Requirement Groups.

<i>Group</i>	<i>Requirements</i>
Display	R1: Render, R2: Place, R3: Allow, R4: Constrain
Data	R5: Position, R10: Enable
Interaction	R7: Create, R8: Define
Results	R6: Transform, R9: Encode

### **3.6.2. Further Analysis: Display, Interaction, Results, Data**

In this sub-section a high level analysis of interfaces is presented, and the following sub-section discusses how the Requirements may be clustered into groups based on results of the analysis.

#### **Display Group**

As was noted in Section 3.4.1 there are certain fundamentals needed for user interfaces, without which user interfaces could not exist or would be very limited in functionality.

The most obvious of these fundamental needs is a display. The display can be visual, haptic, auditory and any other output mode, or fusion of modes. Without a display the user, or users, of a system have no means of receiving communication from the system. If they cannot receive communication then any actions they perform upon the system will never provide any feedback. The system would appear completely non-functioning, effectively a void from which nothing ever comes out.

#### **Interaction Group**

Interactions that can be performed with the system are the second fundamental usually needed in user interfaces. These interactions can take the form of user controlled actions, such as keyboard presses and mouse clicks, or external actions, such as a hardware update of a visual display. Without actions, but with displays, interfaces are only output channels. The user could not cause the interface to respond to them in any manner. Nor would it be possible to alter the interface – because without actions the display would be static and unchanging. This is because actions cause the update of a display. A map could be viewed as a static unchanging display but even then the map was “updated” (created) at a prior time, i.e. the update action is time shifted.

#### **Results Group**

The third fundamental needed are results, which are strongly related to interaction. Results are what happens because of an action, e.g. the displaying of a menu because a menu-bar was selected with a mouse pointer. Results can be thought of as transforms. Interaction and results form the cause and effect cycle in interfaces, i.e. the result of an action is an effect.

#### **Data Group**

Results do not exist in a void, they transform something. Displays also display something. In

both cases it is data. A display delivers data, whether its take the form of images, a sense of touch, audio, etc. Results act upon data, e.g. when clicking an onscreen button they stop the display of one piece of data and start the display of another piece of data. Data is the fourth fundamental need in interfaces. Data can take various forms:

- encoded content, e.g. images
- encoded processes, e.g. transforms and results
- encoded interactions, e.g. regions of a screen that a mouse click should be captured in.

So far in Section 3.6 interfaces have been broken into four broader groupings; these are Display, Interaction, Results and Data. From here on they will be individually referred to as the Display Group, the Interaction Group, the Results Group and the Data Group. As a whole they will be referred to as the Requirement Groups.

#### **3.6.3. From Requirements to Requirement Groups**

Drilling deeper into the Requirement Groups we will refine each one based on the Requirements and related each one to the Requirements (Table 3-2). Analyzing the ten Requirements enables classification of them into the four broader and interdependent Requirement Groups. Refining the Requirements in the context of the Requirement Groups helps shed light on the relationships and dependencies between the various Requirements.

Fulfilling the Requirements within the *Display Group* would result in a basic zoomable information space where the fundamental ZUI operations of zooming and panning a viewport are possible. Of course this would be useless if there was no data within the display. Therefore the *Data Group* in conjunction with the Display Group is required to create a limited but usable ZUI application.

For more complex applications there is a need to monitor user actions and respond to them accordingly. As well as reacting to external occurrences, e.g. updating the display to indicate a file has loaded. Partially realizing this need is possible with the *Interaction Group*, which would enable the creation of a ZUI application that monitors but does not respond to user actions and external occurrences. A complete application necessitates also using the *Results Group* to add and define interface and program behaviours.

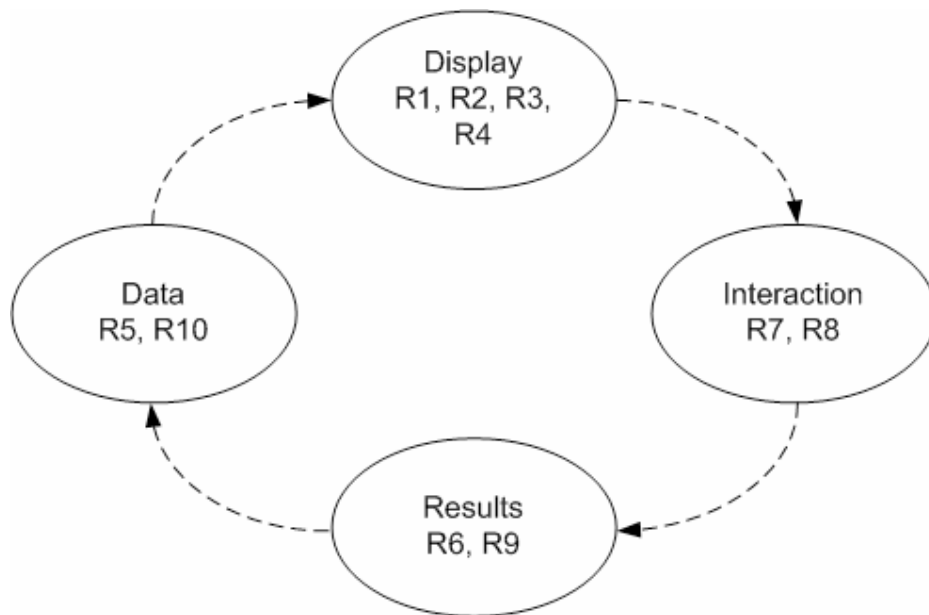


Figure 3-8. Interplay between the Requirement Groups in a Zoomable User Interface.

Within a complete ZUI application the interplay between the groups is, in a simplified form, that which is shown in Figure 3-8. There is a display that enables interaction, which often results in the transformation of the data and that in turn causes display updates.

### **3.7. Conclusions**

In this chapter the Requirements were introduced, defined and explained. They were shown occurring in the prototype application Media Dive.

Creating the Requirements lead to a very specific identification and definition of the properties of ZUIs.

Refining the Requirements in the context of the Requirement Groups helped shed light on the relationships and dependencies between the various Requirements.

# Chapter 4

## 4. Objects, Regions, Relations and Interface Logic

### 4.1. Introduction

This chapter introduces the ORRIL (Objects, Regions, Relations and Interface Logic) framework as a technique for aiding ZUI design and creation. ORRIL makes explicit the data that appears in a zoomable information space, while simultaneously emphasizing the relationships between user actions and transforms of the data.

### 4.2. Why ORRIL

In the previous chapter the Requirements and Requirement Groups were identified, defined and analyzed. While it might be possible to use them as a framework for constructing ZUIs it was felt further analysis would result in a more suitable framework. ORRIL was the result of this.

During the further analysis the aim was to create a *usable* framework for building ZUIs. Whether this aim was achieved would be established by evaluating the framework. Evaluation would involve implementing the framework in a new tool that, if successful, would enable users to rapidly prototype ZUIs. The resulting tool is called Nutmeg and it is covered in depth in subsequent chapters (see Chapter 5 and Chapter 6).

### 4.3. A Usable Framework

What are our criteria for a usable framework? The framework has to be simple, that is having as few components as possible with each component having a clearly defined role. Reducing the number of components means users do not have to keep track of numerous levels and layers of abstraction. Clearly delimiting the role of each component enables users to think about ZUI creation at varying levels of complexity.

The Requirements and Requirement Groups do not meet these criteria. A usable framework means users use the framework to build ZUIs – this may sound obvious but it implies users do not need to be able to alter the first four Requirements. This is because the first four Requirements define the basic environment of a ZUI. So from a user perspective the R1 to R4 Requirements are redundant, or at the very least can be classified into a single unit. Reducing

the Requirements further is covered later in this chapter (see Section 4.6).

Turning to the Requirement Groups we note these too are unusable because they are not specific enough – they are at too high a level of abstraction. A high level of abstraction means they are useful for thinking and describing interfaces at a general level, but not applicable at the level of implementing interfaces.

#### 4.4. ORRIL Defined

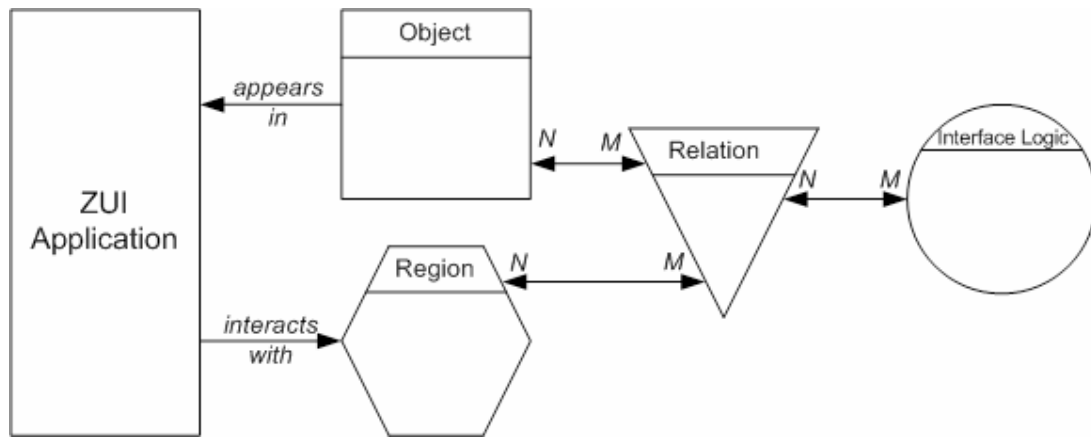


Figure 4-1. The ORRIL model and the relationships between its components.

ORRIL is an abstraction of ZUIs into four basic components (Figure 4-1). This abstraction is a useful framework for understanding ZUIs. From the perspective of an implementer it may be used as an underlying model when designing and implementing a framework for building ZUIs. Alternatively it is useful for modelling and clarifying the processes that occur within a ZUI application.

The four ORRIL components are Objects, Regions, Relations and Interface Logic. Each is defined as follows:



- *Objects* represent a basic perceptual unit within a zoomable information space, e.g. an image, a piece of audio or a block of text.
- *Regions* denote three-dimensional areas where user actions may be captured, e.g. movement of a viewport, continuous updates of a pointer's position, a key press, etc.
- *Relations* define the mappings and associations between Regions, Objects and Interface Logic.

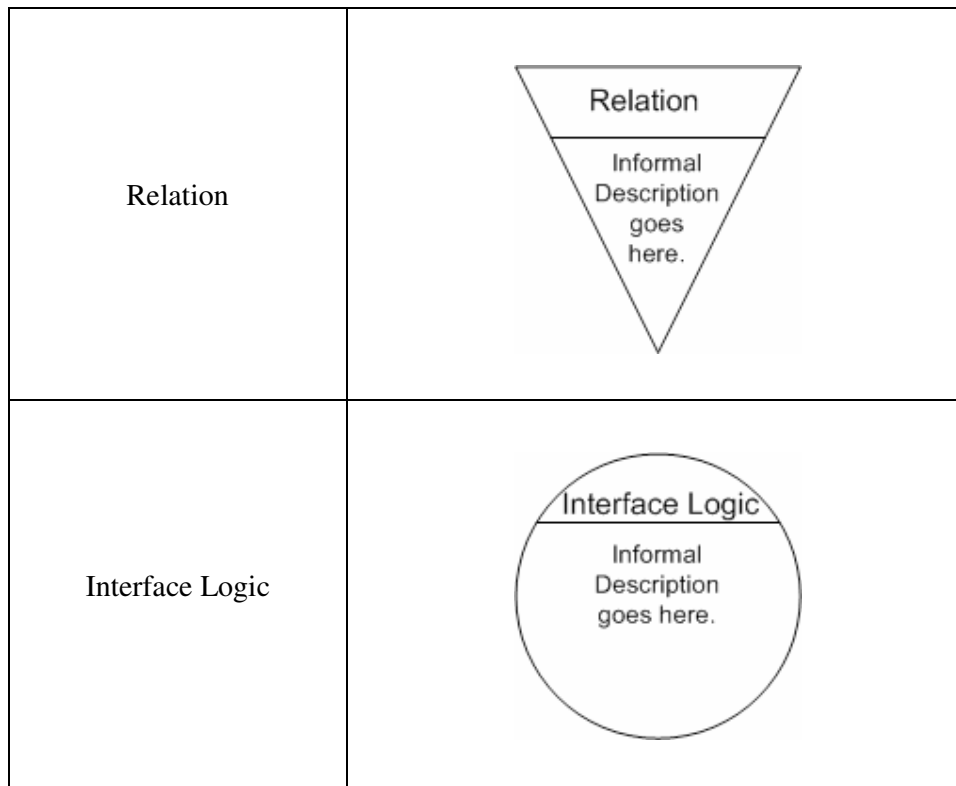
- *Interface Logic* represents transforms that can occur.

ORRIL makes explicit the data that appears in a zoomable information space, while simultaneously emphasizing the relationships between user actions and transforms of the data.

#### 4.5. ORRIL Diagrams

This section describes the informal diagrammatic representations to be used when using ORRIL to describe ZUI interfaces and processes. These diagrams are defined so that there is a consistent diagrammatic representation for ORRIL. In various sections of this thesis these diagrams are used in conjunction with textual descriptions of ORRIL components. The notations for ORRIL diagrams are as follows:

<i>ORRIL Component</i>	<i>Diagrammatic Representation</i>
Object	
Region	



Where it makes sense, according to the ORRIL definition, each of the diagrammatic components can be connected with a line and arrow. The direction of the arrow indicates the relationship between the components, e.g. an arrow pointing from component A to component B implies A leads to B. The nature of the relationship implied by the arrow can be stated by placing a word, in italics, either above or below the line, e.g. *A triggers B*. There may be more than one arrow leading into and out of any component. A number, or N and M, may be near the start and end of an arrow. This indicates the degree/cardinality of relationship between components.

Actions feeding into Regions, such as a Viewport entering a Region, can be depicted by arrows and lines with the nature of the relationship implied by a word in italics. When multiple actions feed into an ORRIL component and multiple actions lead out from the same component the ability to specify which incoming action leads to which outgoing action is needed. This is done by surrounding the incoming action word with () and prefixing it on the outgoing action word (Figure 4-2).

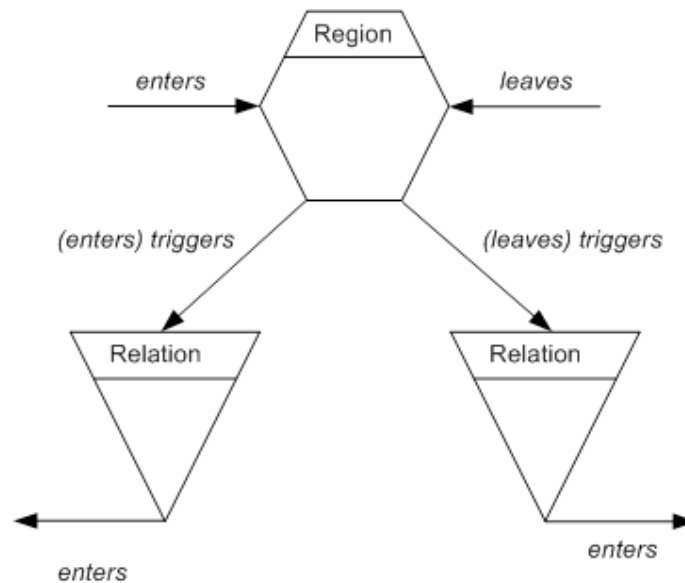


Figure 4-2. Multiple incoming actions leading to multiple outgoing actions.

Diagrammatic representations of the Viewport and a specification of what events can occur, are deliberately left undefined and can be decided upon by the person using ORRIL. The reason the range of acceptable events is not defined is because new events could be invented that interact with a Region. For example a Mouse Button can be *pressed* in a Region, and in the future a Gyroscope could be *tilted* in a Region.

#### 4.6. From Requirements to ORRIL

Briefly returning to the analysis at the end of the last chapter (see Section 3.6.3) it is important to note ORRIL is built on the core assumption that the Display Group is the default environment in which ZUIs exist. This assumption means the very large information space, the display with a viewport and zooming and panning (R1: Render to R4: Constrain) are defaults that are not explicitly captured in ORRIL. In Figure 4-1 the Display Group can be thought of as being part of the ZUI application.

The reason for this assumption is that the R1 to R4 Requirements cannot be altered – they are the foundation blocks for ZUIs. If they were alterable then completely different interfaces types could potentially occur, i.e. variations on 2D and 3D interfaces. While this would be interesting and useful to explore it is a much larger work outside the scope of this thesis.

What follows is an elaboration on how each ORRIL component contributes to satisfying the

Requirements and the Requirement Groups. The following two tables show how the ORRIL components relate to and satisfy the Requirements and the Requirement Groups.

Table 4-1. Mapping between the ORRIL components and the Requirements. <P> means the Requirement following <P> is only partly satisfied/completed.

<b>ORRIL</b>	<b>Requirements</b>
Objects	R5: Position
Regions	R8: Define
Relations	R7: Create, <P> R10: Enable
Interface Logic	R6: Transform, R9: Encode, <P> R10: Enable

Table 4-2. Mapping between the ORRIL components and the Requirement Groups. <P> means the Requirement Groups following <P> are only partly satisfied/completed.

<b>ORRIL</b>	<b>Requirement Groups</b>
Objects	Data Group <P>
Regions	Interaction Group <P>
Relations	Interaction Group <P>
Interface Logic	Results Group, Data Group <P>

#### 4.6.1. Objects

Definition – *Objects* represent a basic perceptual unit within a zoomable information space, e.g. an image, a piece of audio or a block of text.

*Objects* relate directly to *R5: Position*. As was noted in Section 3.4.5 ZUIs need to have

displayable content and/or displayable interface components. *Objects* satisfy this need.

There is a partial relationship with the *Data Group* because *Objects* do not meet *R10: Enable*, i.e. *Objects* can be transformed but they cannot transform other ORRIL components.

#### 4.6.2. Regions

Definition – *Regions* denote three-dimensional areas where user actions may be captured, e.g. movement of a viewport, continuous updates of a pointer's position, a key press, etc.

*Regions* complete the *Requirement R8: Define*. *Regions* act as spatial constraints by limiting whether or not *occurrences*, such as *user actions*, are captured or not – this need was discussed in Section 3.4.8.

An incomplete relationship exists between *Regions* and the *Interaction Group*. The relationship is incomplete because *Regions* cannot be used to create new *mappings* between ORRIL components, i.e. an ORRIL *Relation* needs to be used.

#### 4.6.3. Relations

Definition – *Relations* define the mappings and associations between *Regions*, *Objects* and *Interface Logic*.

*Relations* enable the creation of new ORRIL *mappings* therefore *Relations* satisfy the *R7: Create* and partly complete the *R10: Enable Requirements*. *R10: Enable* is partly satisfied because altering a *Relation* alters a *mapping* but not a *transform*.

The *Interaction Group* is now satisfied as ORRIL's *Relations* complete the *Requirement R7: Create* and similarly ORRIL's *Regions* satisfy *Requirement R8: Define*.

#### 4.6.4. Interface Logic

Definition – *Interface Logic* represents transforms that can occur.

*Interface Logic* directly satisfies both the *R6: Transform* and the *R9: Encode Requirements*, which means it satisfies the *Results Group*.

The remaining unfulfilled aspect of *R10: Enable* is completed by *Interface Logic* because *transforms* can alter (see Section 3.4.9 and Section 3.4.10) *transforms* and *mappings*. This completes the *Data Group*.

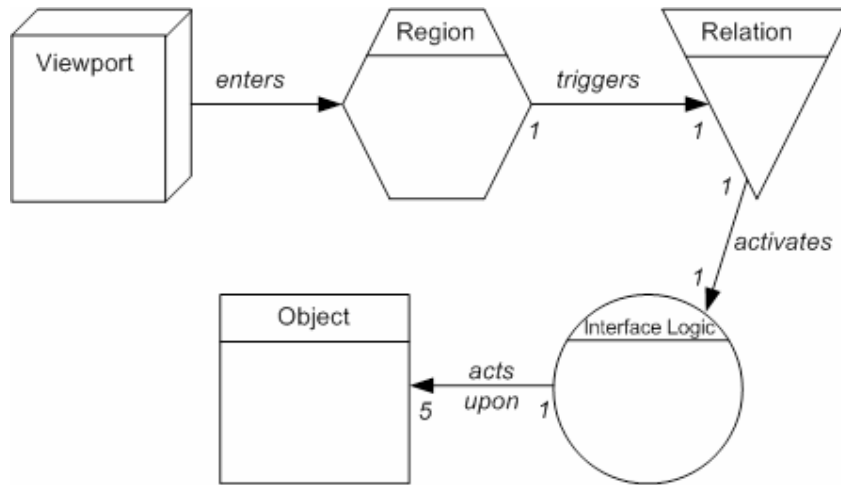


Figure 4-3. Process of zooming towards five image Objects in ORRIL.

#### 4.6.5. Degree of Relationship Between the Components

ORRIL’s Relation acts as the glue that binds ORRIL’s components (see Appendix B) into a range of relationships. The relationships between the components are often N:M.

There is a need for N:M relationships, rather than 1:M or N:1 or any limited variation thereof. An example of an N:M mapping required in a ZUI is when a user has control of a ZUI viewport. Imagine if the viewport was zoomed towards what were initially five little dots (small images) that turn into five larger and richer Objects, such as different images and blocks of text.

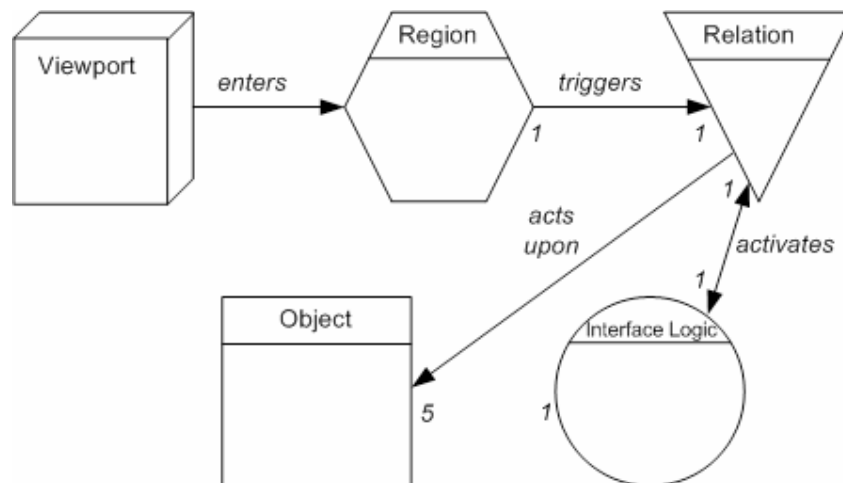


Figure 4-4. Zoom with a 1:1 relationship between Interface Logic and a Relation, but a 1:5 relationship between a Relation and Objects.

This zoom process (Figure 4-3) causes the viewport to enter a Region, which triggers an ‘enter’ event that causes Interface Logic to run by means of a Relation. This Interface Logic acts upon five Objects, one for each image. There is a 1:5 relationship between Interface Logic and Objects. Interface Logic was bound to always act upon the same Objects. This need not have been the case. The Interface Logic could have been more generic and written to act upon any kind of image Object. For example (Figure 4-4), the Interface Logic could have been associated with a Relation, and the Relation could be associated with five Objects, thus indirectly relating Interface Logic to five objects.

### 4.7. Examples of ORRIL

This section gives three examples of using ORRIL. This is done to help illustrate and clarify how the ORRIL components work together to describe ZUIs.

#### 4.7.1. Using Only Objects

If a user wants to quickly create a ZUI, or is unfamiliar with them, they could think only in terms of Objects. Each Object could be associated with a single image. By placing Objects within a zoomable information space they would have control over where the images appear, thus creating a basic ZUI application.

#### 4.7.2. Activating a Song

The following is a more complex example where all four ORRIL components are used. In Section 3.5 Media Dive was analyzed from the perspective of the Requirements – here it will be presented in the ORRIL framework (Figure 4-5).

Media Dive requires that songs and images are placed within a large zoomable information space. With ORRIL this is done

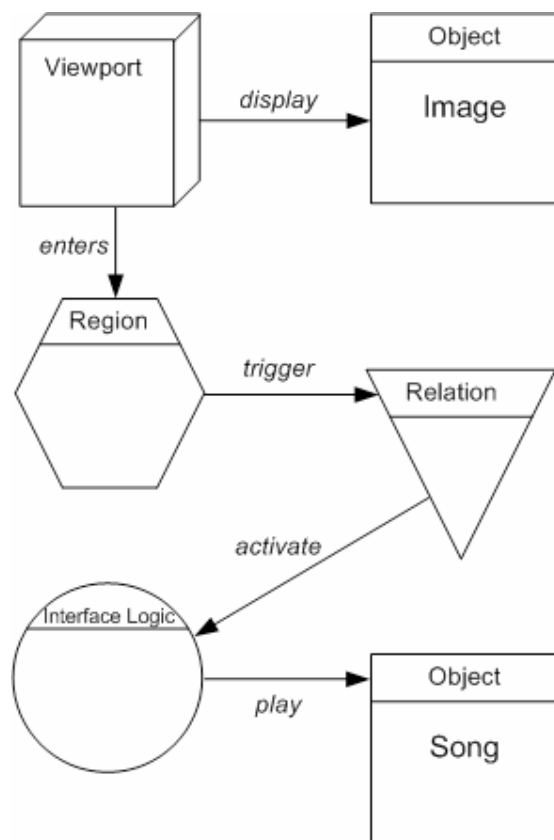


Figure 4-5. Playing a song in Media Dive and showing an image.

by associating each song and each image with an Object. Each Object is then given a specific location within the zoomable information space. If the viewport displays an Object location then the Object's image will be displayed in the viewport.

When a Media Dive user zooms in to an image the song associated with the image begins to play. This is done by creating one Region for each image. Each Region is given a location with the result that a Region is placed in front of each image Object. When the viewport enters a Region a user event occurs that eventually leads to the song (Object) starting.

The user event will lead to nothing if it is not linked to a transform function, such as starting or stopping the song. A Relation maps the user event to a transform function. The transform function is encoded in the Interface Logic.

A Region captures the user event. The user event is mapped via a Relation to Interface Logic. The Interface Logic is a function that acts upon a song Object. The function acting on the song causes it to start playing. The music stops when the same Region detects a user 'leave

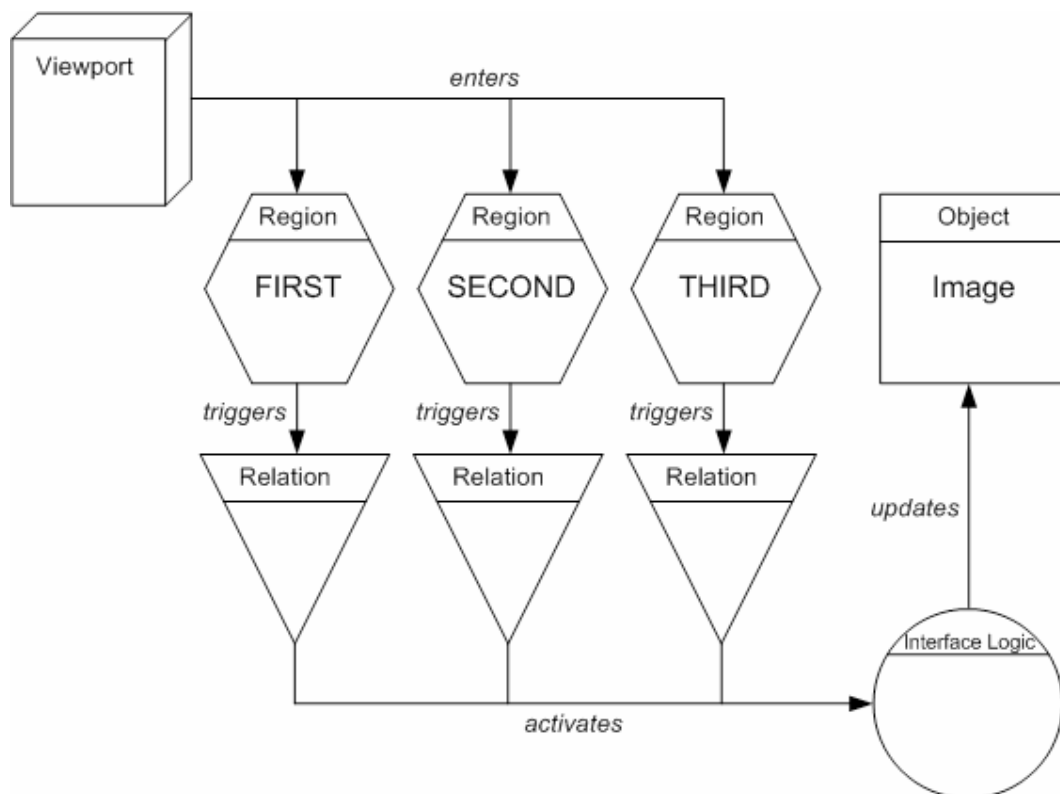


Figure 4-6. ORRIL representation of semantic zooming with multiple updates to the same image Object.

event'. The user 'leave event' is mapped to a function for stopping the music playing.

#### **4.7.3. Semantic Zoom**

The final example (Figure 4-6) covers using ORRIL to outline the process of semantic zooming. In this case ORRIL is used to explain what occurs when zooming towards a single image. As the zoom occurs the image continuously gets bigger and the content of the image is updated in discreet steps. Figure 4-6 shows three distinct Regions that are in front of the image Object. When the user controlled viewport enters Region FIRST the image Object is updated by the activated Interface Logic. Transitioning from Region FIRST to Region SECOND causes another activation of the Interface Logic, which again updates the image. This also occurs when transitioning from Region SECOND to Region THIRD.

#### **4.8. Conclusions**

This chapter introduced ORRIL and showed how it was derived from the Requirements and Requirement Groups.

ORRIL was clearly defined as a usable model and a diagrammatic representation of it was presented.

The use of ORRIL in textual and diagram forms was demonstrated for describing a range of commonly required ZUI actions – this contributed towards understanding whether ORRIL is usable for constructing ZUIs.

The following three chapters cover the evaluation of ORRIL by implementing it as a new tool and a markup language and establishing whether the implemented form of ORRIL simplifies the process of creating ZUIs.

# Chapter 5

## 5. Nutmeg: A Test Implementation of ORRIL

### 5.1. Introduction

How successful and useful is ORRIL? This chapter discusses the issues and processes involved with using ORRIL as the basis for Nutmeg. Nutmeg, created as part of this research, is an experimental tool for rapidly prototyping medium to high fidelity zoomable user interfaces. The rationale for the creation of Nutmeg was to evaluate ORRIL by doing a test implementation of it. This test implementation helps establish what range and style of ZUIs could be implemented with ORRIL.

In other words ORRIL was converted from an abstract model into a test implementation to establish and evaluate the validity and expressiveness of ORRIL for defining and creating ZUIs.

### 5.2. A New Tool

Nutmeg is a new tool for rapidly prototyping medium to high-fidelity [50] multi-modal zoomable user interfaces. Prior to Nutmeg nearly all the tools for constructing ZUIs were low-level libraries [5, 6, 7, 17, 22, 29, 41] with clearly defined APIs that required a considerable degree of technical skills to use. The most obvious benefit of Nutmeg is that reduces the time and skills needed for creating ZUIs.

An important design criterion in the creation of Nutmeg is that it should enable people to start building simple ZUIs and proceed to creating richer ZUIs as their skills, understanding and knowledge increase.

For example, with the Hyper-Text Markup Language (HTML) a non-technical individual can simply create a web page by only using the IMG tag with blocks of text. This does not preclude the ability to use more complex HTML tags and create more complex web pages as user's skills and knowledge increase, and/or tools become available that simplify the process of creating complex web pages.

A secondary design criterion was that Nutmeg should not concern itself with internal details of the media objects it presents; instead the focus is on loading externally generated media

into Nutmeg and using that media to create the ZUI interfaces. This decision was motivated by the aim of keeping Nutmeg focused on rapid prototyping. If Nutmeg had a byte orientated structure, that is if prototype ZUIs could have their images and audio sources altered at a byte by byte level, then the amount of details that would have to be taken care of in implementing a ZUI would greatly increase, e.g. when a ZUI implementer moves an image, do they move the image byte by byte or do they move the image as a single unit? There is a trade off here, as certain kinds of operations cannot be easily performed in ZUIs depicted by Nutmeg, e.g. giving an on screen image a blue tint involves loading a tinted copy of the image and replacing the untinted image, rather than processing each byte of the existing image and performing some Boolean operations to add the tint. The effect of this trade off is that it places some limits on the style and range of ZUIs that can be created with Nutmeg.

### **5.3. Overall Structure**

Nutmeg functions somewhat like a web browser. In the next chapter, Chapter 6, ORRIL is converted into a markup language called the Zoomable Interface Markup Language (ZIML). Nutmeg reads in ZIML and parses and stores it in a Document-Object Model (DOM) [55]. The information in the DOM is used to layout and render an interactive ZUI on display devices. The display can be visual and/or auditory. Interface and programming logic can be added to the ZUIs via a range of familiar scripting languages. The media that appears in a ZUI, and is referenced by the ZIML, may be images, audio and blocks of text. A diverse range of widely used image and audio formats are supported. The images and audio can be stored on remote machines and accessed via standard network protocols.

### **5.4. Capabilities and Functionality**

Table 5-1 shows the main capabilities and functionality of Nutmeg. Following this is an explanation of each key aspect of Nutmeg. While it would have been possible to delve much deeper into each aspect it was felt that this would result in the introduction of many irrelevant technical details that would distract from evaluating the validity of ORRIL and potential benefits of Nutmeg.

Table 5-1. The important aspects of Nutmeg and the capabilities of each aspect.

<i>Supported</i>	<i>Capability</i>
Basic Environment	ZUI window controllable with a pointer.
Operating Systems	Any platform that supports Java applets and/or Java applications.
Markup Language	ZIML (Zoomable Interface Markup Language)
Reading Files	Local file system, remote network servers via HTTP
Images	Jpeg, gif, png, bmp, etc.
Audio Sources	Mp3, ogg, au, wav, etc.
Scripting Languages	Java as a script, JavaScript, TCL, Python, PerlScript, VBScript, Rexx, etc.

#### 5.4.1. Basic ZUI Environment

As was previously discussed (see Section 3.6) ORRIL is built on the core assumption that the Display Group is the default environment in which ZUIs exist. This assumption means the very large information space, the display with a viewport and zooming and panning are defaults not explicitly captured in ORRIL. That is each component of ORRIL exists in the context of a standard ZUI. Note this also means that zooming and panning, as controlled by the user, such as with a pointer, is separate from ORRIL.

Nutmeg takes this core assumption and uses it to provide a basic environment in which all ZUIs exist. By default the basic environment a Nutmeg user is presented with is a window into a large zoomable information space that can be navigated with a pointer.

#### 5.4.2. Applet and Application

Nutmeg is written in Java and all efforts were made to make sure it functioned as both a cross

platform application and as a downloadable applet that can be embedded in a web page. Differences between how applets and applications are treated by Java's security model means using Nutmeg as an applet places some impositions on how ZUIs can be defined in ZIML. One example of an imposition is images can only come from where the Nutmeg applet is downloaded from. In applications the images can come from any location, such as a local file system or a remote web server.

#### **5.4.3. ZIML DOM**

Central to Nutmeg is the ZIML Document-Object Model (DOM) [55]. This DOM stores all the details and state information about a ZUI. For example, the DOM stores details about what media appears in a ZUI, where the media should be spatially located and whether the media is currently shown in the display. The DOM is populated with ZIML, which is a custom developed eXtensible Markup Language (XML) [54]. When Nutmeg starts, a ZIML file is passed to it. This file may be locally stored or accessible via HTTP.

#### **5.4.4. Multiple Audio Formats and Streams**

More than five different audio types are supported, such as Mp3s and Ogg files, including audio streamed from locations on networks and the Internet. Multiple audio sources, up to twenty, can play simultaneously. Supporting multiple simultaneous audio sources is important because it enables creators of ZUIs to experiment with audio in ZUIs in a way that, prior to Nutmeg, was not possible without a considerable amount of development, e.g. what benefits are there to using multiple audio sources in conjunction with showing large datasets in ZUIs?

As new audio formats arise they can easily be added to Nutmeg, due to the underlying architecture used, i.e. add a new library with the appropriate API to the Nutmeg directory. In this prototype of Nutmeg the starting and stopping of audio playback and the volume are controllable but full audio spatialization is not supported. Extending Nutmeg for full audio spatialization is possible as there are a number of Java libraries that enable audio spatialization.

#### **5.4.5. Numerous Image Formats**

Numerous image formats are supported by Nutmeg, e.g. Jpeg, gif, png, bmp, etc. These images can be stored on local file systems and remotely at network and Internet locations. Images must be static. Animated gifs or similar formats are not supported. Network

connections for downloading and displaying images occur via HTTP requests to web servers. Images must be allocated X, Y and Z coordinates if they are to appear in Nutmeg – these coordinates can be changed. Control over whether the images are shown at any moment in time is possible by dynamically adding and removing the images.

#### **5.4.6. Scripting Languages**

ZUIs created in Nutmeg can be made responsive to user actions and external events. That is ZUIs in Nutmeg can be much more than static collections of images and audio pieces. Interface and programming logic can be encoded in a wide range of scripting languages. Each unit of logic may be a collection of methods and functions organized in a coherent manner to carry out specific tasks, e.g. increase the volume of a piece of audio as a person zooms towards a specific image.

If required, multiple scripting languages can simultaneously run while accessing and altering a ZUI. Each instance of a script lives in its own thread until its task is completed; this prevents ZUI applications from blocking while waiting for interface updates, etc. Scripts can be written in JavaScript, Python, Tcl, Rexx, PerlScript, VBScript and numerous other languages. A benefit of this diversity in scripting capabilities is that a wide range of skill sets can be brought to bear on the task of creating ZUIs. Scripts may be stored locally or they may come from remote servers.

Scripts have full access to the DOM and can completely alter most aspects of it. Updates to the DOM are not committed instantly. Instead there is a basic transaction model for commits akin to that which is commonly used in SQL databases. This is done to prevent partially complete updates getting shown to a user, e.g. if an image is getting moved around the screen along the X and Y axis, and if only the X axis information has been updated, the image should not be moved until the Y axis information has also been updated.

One other noteworthy aspect of the scripts is they are late binding, or more particularly they are on-demand binding. This means that only when a script is activated does it get loaded and evaluated. An obvious benefit of this is that it lowers the memory footprint required by Nutmeg. A less obvious, but more important, benefit is that scripts can radically alter the behaviour of a running ZUI, and not just its look and feel. Of course a disadvantage of this is the added time taken to access and load a script – but this could be resolved with a well-designed caching architecture.

### 5.5. Design and Implementation

Nutmeg is written in Java with a hierarchical multi-threaded architecture. There are multiple managers and independent cooperating threads that have responsibility for different aspects of Nutmeg. Cooperation is organized via inter-thread method calls with some message passing.

Drilling a little deeper into the architecture of Nutmeg it is noted that each manager has a specific domain of responsibility. There are eight managers and one top-level manager, i.e. Manager, DOM Manager, Object Manager, Region Manager, Relation Manager, Logic Manager, Event Manager, Interface Manager and Audio Manager. Most managers do not have sub-managers, except in the case of the audio manager.

A multi-threaded approach helps prevent blocking on interface updates and provides a measure of scalability, i.e. running Nutmeg on a JVM (Java Virtual Machine) capable of taking advantage of a multiprocessor computer could enable threads to be assigned to different processors. Scalability is especially important because ZUIs may be used to depict, explore and enhance interaction with large data sets, including multimedia data sets.

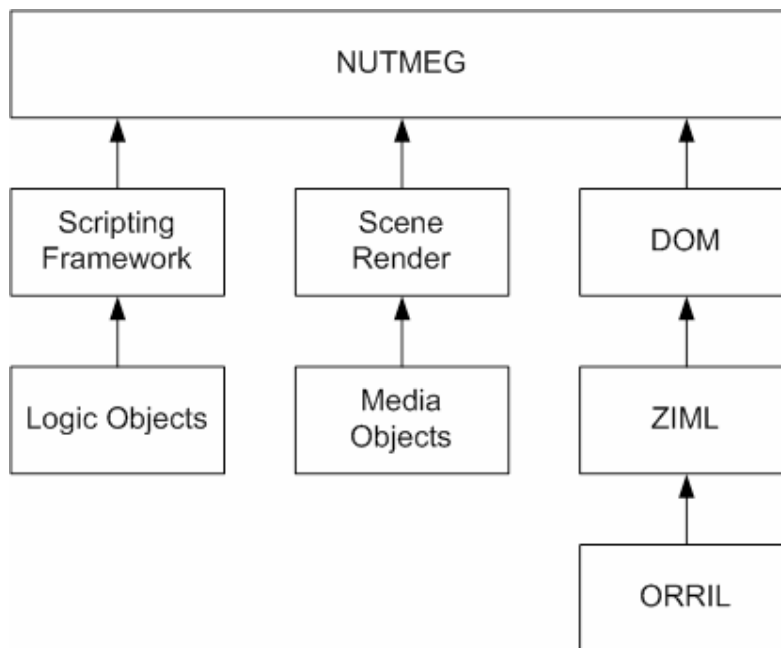


Figure 5-1. Simplified representation of the hierarchical structure of Nutmeg.

Looking at Figure 5-1 we see a simplified representation of the hierarchical structure of Nutmeg. This simplification shows the three main aspects of Nutmeg – the scripting

framework, the scene render and the DOM.

In many ways the DOM is the core of Nutmeg, which the scripting framework and scene render constantly depend upon and interact with. For example, the content of the DOM tells the scene render what user input it should monitor and what scripted actions, in the scripting framework, to trigger. The DOM stores most of the information about individual ZUI components, e.g. an image's location and whether it should be shown in a ZUI. ZIML files, which are loaded into the DOM, specify the layout, appearance and behaviours of ZUIs depicted by Nutmeg. Scripts (see Section 5.4.6) can alter any part of the DOM. Part of the development of Nutmeg included designing and implementing an API that scripts can use to alter the DOM. Changes to the contents of the DOM alter the displays and behaviours of ZUIs.

The exact structure and contents of the DOM is based on ZIML, which in turn is based on ORRIL. Chapter 6 elaborates upon the structure and role of ZIML and its relationship to ORRIL.

The scripting framework takes care of loading interface logic scripts and interpreting them in a range of scripting languages. Interface logic scripts are files, stored either locally or remotely. Depending on the filename of a script, the scripting manager loads an interpreter that can run the script. Multiple interpreters can be loaded and running at once. This feature enables multiple scripting languages to be in concurrent use. The Bean Scripting Framework (BSF) [23], from IBM's research lab, enables the processing of the script, while the interpreters come from many sources and new ones can easily be added as new scripting languages are developed.

As a whole the scene render creates and manages a visual display, an auditory display and monitors user inputs. Media Objects are media files and streams, such as MP3s and Jpegs. The scene render knows what media Objects to load and where to locate them based on the contents of the DOM. The visual part of the scene render takes care of showing users a visual zoomable information space and enabling users to interact with the scene. This was built with Piccolo [5], a low-level library for creating visual ZUIs from University of Maryland's Human Computer Interaction Lab. Since little research has been done on audio in ZUIs some low-level development work had to be completed to enable multiple streams of audio in Nutmeg ZUIs. Low-level audio format handling, decoding and playing is done with the JavaZoom's JLayer [26].

At least twelve different low-level libraries were used in the construction of Nutmeg. So many libraries had to be used because creating a tool like Nutmeg without externally developed libraries would have been an absolutely mammoth task. There are libraries for reading different audio file formats, for interpreting numerous scripting languages, for reading in and parsing an XML variation, and libraries for handling zoomable scene rendering and taking in user input. Some of these libraries were not thread aware so special care had to be taken in how they were integrated into Nutmeg.

## **5.6. Conclusions**

This chapter explained the basis for a new tool for rapidly prototyping zoomable user interfaces. The focus on simplicity and usability was emphasized and the motivation for certain design decisions was touched upon.

Nutmeg is a powerful new tool that potentially enables users to quickly build and develop ZUIs in a manner that otherwise requires considerable programming and development skills.

# Chapter 6

## 6. From ORRIL to ZIML

### **6.1. Introduction**

In a previous chapter ORRIL was covered in depth – this section builds on that by explaining how ZIML derives from and is based upon ORRIL. ORRIL consists of four core components: Objects, Regions, Relations and Interface Logic. Implementing ORRIL as a usable framework meant a close examination of each component had to occur. Exactly what are the properties of the components? Are four components enough for creating a usable markup language for describing ZUIs?

### **6.2. Converting from ORRIL to ZIML**

Converting from ORRIL to ZIML involved establishing the implicit attributes of each ORRIL component. These attributes become relevant when using ORRIL for an implementation framework. They become relevant then because the attributes are often implementation dependent. Those attributes that are not implementation dependent are implied by each ORRIL component definition. For example, from the definition of an Object we know that an Object must have a three-dimensional spatial location because it exists in a zoomable information space. On the other hand the answer to the question “How should an Object be associated with a piece of piece of media?” can vary greatly, depending on specific implementation choices, e.g. are the image bytes stored in the markup language description of Objects, or are images stored in separate files and referenced via file paths?

### **6.3. Separation of Content, Layout and Logic**

As has been noted elsewhere an important aspect of Nutmeg and ORRIL is that they should be simple to use and easy to understand. To achieve this a web browser, HTML and Model-View-Controller influenced approach was adopted as part of the design motivation for ZIML. This end result of this meant an emphasis was placed on maintaining a separation between 1, 2 and 3:

1. Logic (interface and program logic)
2. Interface (interface description and layout information)

3. Media (the media that appears in the interfaces).

When it was deemed much more work than was reasonable for the development of the prototype (Nutmeg) the separation was not as strictly enforced. The separation is not rigorously enforced because blocks of text can be included in ZIML Objects, rather than having the text stored in separate files.

Using a HTML-like markup language makes things simpler for two reasons. Firstly, many people have transferable skills in HTML development. HTML enforces some separation between layout information and media information, i.e. images are stored in separate files. Secondly, maintaining a clean separation enables users to develop scalable skills. Developing scalable skills is feasible because a user could initially focus on understanding how to layout an interface and as their skills increase they could then begin to implement interface and program logic.

#### **6.4. Objects**

Objects are any kind of media that exists within a zoomable information space display. An implicit attribute of an Object is its spatial location; this is the <location> tag in Example 6-1. Media content is associated with an Object in the <content> tag. Each Object requires an ID, so interface logic can select what Objects to act upon. The <info> tag was added so ZIML author comments could be associated with each Object. The <uri> tag specifies where external resources can be found. Nutmeg's URI's are partially based on the Uniform Resource Identifiers defined in RFC2396 [8]. Below is an example of an Object, with an ID of "a6" that loads a Jpeg image from a website and displays it at location 0, -400, 1.

*Example 6-1. A ZIML Object which sets up an image in the zoomable information space.*

```
<object id="a6">
  <info>Display a cute picture</info>
  <content type="image/jpeg">
    <uri>http://www.somewhere.none/imageA.jpg</uri>
  </content>
  <location>
    <x>0</x>
```

```
<y>-400</y>
<z>1</z>
</location>
</object>
```

## 6.5. Regions

Regions define areas where user actions may be captured. Regions are also like Objects in that they require an identifier so they can be referenced within the ZIML – this is the id. In Example 6-2 a <location> tag is also used to spatially position a Region. The <shape> tag is used to define the three-dimensional shape of Regions where user actions may be captured. The <fulc\_\*> tags specify the front upper left corner of the three dimensional rectangle and the <blrc\_\*> tags specify the back lower right corner of the rectangle. Different types of shapes could be specified but in this version of Nutmeg only support for rectangular areas was implemented.

Potentially, a Region could have been implemented as a special type of Object, a sub-class of Object. This blurring of boundaries between Regions and Objects might have led to ambiguity over the exact roles of each, so it was avoided.

In the following example a rectangular Region with the ID of “r2” is spatially located at position 200, 200, 0. It is worth noting that the numbers in <location> and <shape> are independent. The numbers in <shape> define the shape of a Region independently of its position as specified by the <location> tag.

*Example 6-2. A ZIML Region that is rectangular.*

```
<region id="r2">
  <info>Capture what user does when viewing A6</info>
  <shape type="rectangle">
    <fulc_x>40</fulc_x>
    <fulc_y>40</fulc_y>
    <fulc_z>7000</fulc_z>
```

```
<blrc_x>-40</blrc_x>
<blrc_y>-40</blrc_y>
<blrc_z>750</blrc_z>
</shape>
<location>
  <x>200</x>
  <y>200</y>
  <z>0</z>
</location>
</region>
```

## 6.6. Relations

Relations define mappings and associations between Regions, Objects and Interface Logic. In many ways a Relation acts like a filter – it binds Region events to units of Interface Logic and may also specify what Objects are to be acted upon by the Interface Logic.

For example, in the Relation “or1”, shown on the following page (Example 6-3), if there is an enter <event> in the “r2” <regions> then the JavaScript file “logic/image\_on.js” is run and told to act upon the <objects> with identifiers of “a6”.

The conversation of an ORRIL Relation to a ZIML Relation is less straightforward than with converting Objects and Regions. Should Regions, rather than Relations, be used to store information about what events to watch for?

The motivation for having Relations store event information is that Relations bind Regions, Objects and Interface Logic into complex groupings. Specifically Relations can encode relationship information between Regions, which capture user actions, and Interface Logic. Events function as a way of specifying the kind of relationship between the Regions and Interface Logic, e.g. only let a mouse click trigger Interface Logic but do not let a key press trigger the Interface Logic.

Returning to the example Relation “or1” each Relation has an ID. The <objects> tag can reference multiple Objects, e.g. <objects ids=”a6, b3”>. The <regions> tag functions in an

equivalent manner by enabling the referencing of multiple Regions. This fulfils the N:M mapping between Regions and Objects described in Section 4.6.5. There can be multiple <event> tags, each of which can be reference from scripts via an ID. A number of <event> types are supported and support for more event types could be added.

*Example 6-3. A ZIML Relation linked to events that can trigger the referenced JavaScript.*

```
<relation id="or1">
  <info>Change the A6 image</info>
  <objects ids="a6"/>
  <regions ids="r2"/>
  <event id="e1" type="enter">
    <uri>logic/image_on.js</uri>
  </event>
  <event id="e2" type="leave">
    <uri>http://www.somewhere.ie/logic/image_off.js</uri>
  </event>
  <event id="e3" type="move">
    <uri>logic/jiggle_image.js</uri>
  </event>
</relation>
```

## 6.7. Interface Logic

Interface Logic does not have a unique tag; it is referenced via the <uri> sub-tag of an <event> tag in a Relation. This was done to maintain simplicity and usability of the ZIML. Adding an <interface logic> tag was an unnecessary complication because it would introduce another tag and layer of abstraction for the user to keep track of. Realistically all the <interface logic> tag would have done is point to the location of the logic, i.e. it would have wrapped a <uri>. The Interface Logic could have been implemented as a special class of an Object, but the same argument put forward against doing this for Regions holds true (see Section 6.5).

Below is a short sample (Example 6-4) Interface Logic script (called imageFlick.js). The script is written in JavaScript. It repeatedly switches between two images associated with the “a6” Object (see Section 6.4). First it sets up a variable called currImageUri. Then it assigns the current value of the <uri> tag in the “a6” Object to the variable. Next the variable is checked to see what image is associated with the “a6” Object. Based on the results of the check it sets the “a6” <uri> to a different image. After that Nutmeg is told the DOM has been changed so a display update should occur.

*Example 6-4. JavaScript code that can change what image is displayed in Nutmeg.*

```
var currImageUri;

currImageUri = ziml().object("a6").uri();

if(currImageUri.value("images/imageA.jpg") == true)
{
    ziml().object("a6").uri("images/imageB.gif");
}
else
{
    ziml().object("a6").uri("images/imageA.jpg");
}

ziml().update();
```

## **6.8. An Example of ZIML**

Now that all the ORRIL components are converted to ZIML they will be put together to create a ZUI. This ZUI illustrates how the components coexist, relate and work together.

The ZUI application is very simple; it starts showing picture A in a window (Figure 6-1, Screenshot 1) on the user’s desktop. When a user zooms closer than a fixed distance to the picture the picture is replaced with picture B (Figure 6-1, Screenshot 2). If the user zooms

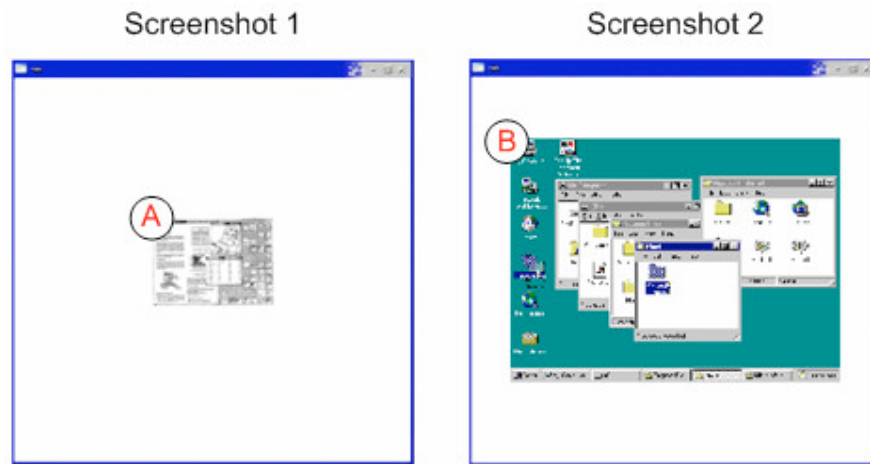


Figure 6-1. Screenshot 1 and 2 show a window alternating between image A and image B.

more than a set distance away from the picture the picture is replaced with picture A. Zooming back and forth will alternate between picture A and picture B indefinitely.

The ZIML for this demonstration ZUI application is below. Comments are in the ZIML to explain how it works. The comments about the code can be found in the `<info>` tag. Note: the JavaScript referenced in the `<relation>` as “imageFlick.js” is the code used in Example 6-4. This ZUI application works by placing a Region in front of an image, if the viewport enters or leaves the Region then the Interface Logic is run which updates the “a6” Object with a different image.

Example 6-5. ZIML for alternating what image is shown to a user depending on how the user zooms.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ziml xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="ziml.xsd">
  <object id="a6">
    <info>Default image to display</info>
    <content type="image/jpeg">
      <uri>images/imageA.jpg</uri>
    </content>
    <location>
      <x>1</x>
      <y>1</y>
    </location>
  </object>
</ziml>
```

```
<z>50</z>

</location>

</object>

<region id="r2">
  <info>Capture what user does when viewing A6</info>
  <shape type="rectangle">
    <fulc_x>1</fulc_x>
    <fulc_y>1</fulc_y>
    <fulc_z>1</fulc_z>
    <blrc_x>100</blrc_x>
    <blrc_y>100</blrc_y>
    <blrc_z>50</blrc_z>
  </shape>
  <location>
    <x>1</x>
    <y>1</y>
    <z>1</z>
  </location>
</region>

<relation id="or1">
  <info>Change the A6 image</info>
  <objects ids="a6"/>
  <regions ids="r2"/>
  <event id="e1" type="enter">
    <uri>imageFlick.js</uri>
  </event>
  <event id="e2" type="leave">
    <uri>imageFlick.js</uri>
  </event>
</relation>
```

```
</relation>
</ziml>
```

### **6.9. ZIML's XSD**

It is worth noting that because ZIML is an XML variant an XSD (XML Schema Definition) was created while designing the markup language. An XSD formally defines a markup language. While it is acceptable to say ORRIL was converted to ZIML, it is more accurate to say ORRIL was converted to an XSD that defines ZIML. An XSD does not have to be created but XML parsers can be told to valid against an XSD to ensure correctness in how a markup language is used. Appendix A lists the XSD – details on reading the XSD can be found at W3C.org.

### **6.10. Conclusions**

Converting ORRIL into a usable framework is easily done. Directly mapping ORRIL into ZIML is readily achieved with no unnecessary new layers of abstraction required. Certain choices were arbitrary when converting from ORRIL to ZIML but these were limited in nature.

A case was made for why maintaining a strong separation between layout, media and logic is in the user's interest.

An initial evaluation was done. This showed that ZIML and indirectly ORRIL are usable for creating ZUIs. Carrying out further "use evaluation" is the focus of the following chapters.

# Chapter 7

## 7. Prototyping Applications with Nutmeg

### 7.1. Introduction

How usable is Nutmeg for creating ZUIs? By examining this question we indirectly evaluate the capabilities of ORRIL. This chapter covers using Nutmeg for constructing ZUI applications and for implementing core ZUI interaction techniques. In particular the differences in two approaches to implementing an application called Media Dive are highlighted and examined.

### 7.2. Media Dive

Media Dive (Figure 7-1) is a prototype audio-visual ZUI application that, as part of this work, was designed and implemented for browsing large audio collections, such as songs.

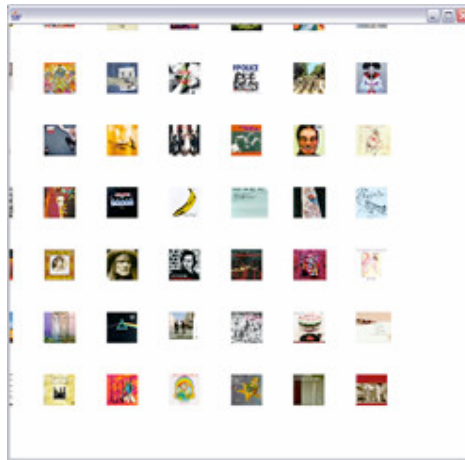


Figure 7-1. Screenshot of Media Dive.

The visual interface displays many images, each of which is associated with a song. A user browses the songs by panning and zooming. If a user zooms in to an image the associated song will begin to play and if a user zooms out the song will stop.

#### 7.2.1. Functionality of Media Dive

The functionality (Figure 7-2) of Media Dive is as follows:

- a user is presented with a visual GUI window (a visual viewport) on a display device,

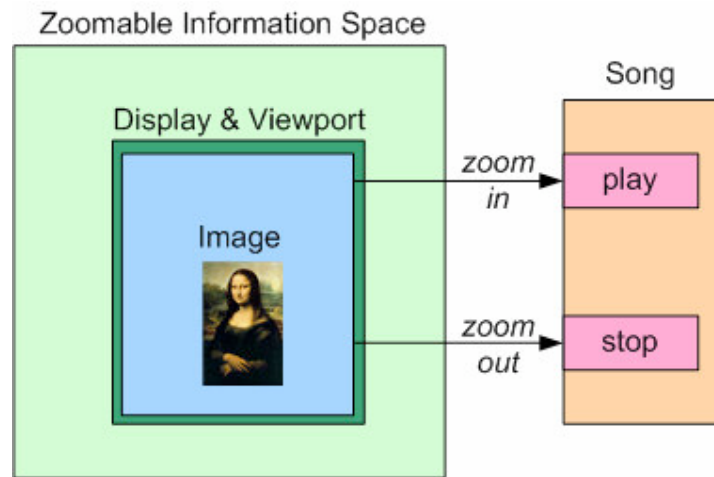


Figure 7-2. Process that occurs within Media Dive.

this window can be resized (Figure 7-1)

- graphical images are displayed in the window
- images are laid out in a grid, with a fixed distance between each image
- each image is associated with a piece of audio
- images can be zoomed in and out, along the Z axis
- zooming occurs on all images equally, i.e. zooming does not occur on a single image while the surrounding images do not increase or decrease in scale
- images can be panned left and right, up and down, along the X and Y axis
- panning occurs on all images equally
- a pointer, such as a mouse, is used to control zooming and panning
- the pointer appears on the display as an arrow
- to pan a user holds down the first pointer button and then moves the pointer in the direction they desire the pan to occur in, i.e. move the pointer left and the images all move left, etc.
- the rate of panning is proportional to how fast the pointer is moved

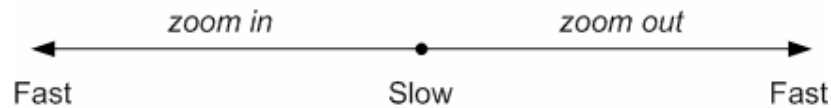


Figure 7-3. Moving the pointer left and right controls zooming in and out.

- to zoom a user holds down the second pointer button and moves the pointer left to zoom in and moves the pointer right to zoom out
- zooming occurs towards or away from a point, this point is the X and Y coordinate of the pointer each time zooming is activated
- zooming can occur at varying rates, from very slow to very fast
- the rate of zooming is proportional to the distance the pointer is moved either left or right from the point the zooming was activated at (Figure 7-3)
- zooming is continuous while the pointer is away from the point zooming was activated at
- zooming is stopped by releasing the second pointer button or by moving the pointer reasonably close to the X and Y coordinate the zooming was activated at
- when the pointer moves outside the graphical window the last state and action is maintained and continued, i.e. if zooming out was occurring then it will keep doing so, if panning left was occurring then it will continue indefinitely
- a piece of audio associated with an image stops and starts based on proximity activation, i.e. when an image is scaled up above a certain size, where it appears large in the display window, the audio will start, when the image is scaled down below a certain size the audio will stop
- audio starts playing at a fixed volume, with the volume increasing or decreasing to maximum and minimum values as a function of image scale, i.e. the bigger the image in the display the louder the associated audio, and the reverse is true for decreasing audio volume .

### 7.2.2. Using Media Dive

Media Dive appears on the screen as a window (Figure 7-1). There are forty-nine small

images centred in the window. At this point the user can decide how to interact with the zoomable user interface.



Figure 7-4. Three different screenshots of what is seen in the Media Dive viewport at different scales.

For example, the user could explore the audio associated with each song by moving the onscreen pointer over an image, then pressing and holding the second pointer button and moving the pointer a small amount to the left to slowly zoom towards the image (Figure 7-4). This causes the display to undergo continuous updates such that the images grow larger (from Figure 7-4 Position 1 to Figure 7-4 Position 3). Once the scale of the image has increased beyond a certain size the audio associated with the image plays. If the user zooms out from the image the audio stops.

Potential issues with the Media Dive interface are that a user could zoom in to a point between the images and effectively get lost, i.e. they would have no frame of reference. They are not told how far they have zoomed, nor would they be able to tell the effect of their current actions – there is only a white background onscreen that looks exactly the same when panning or zooming. These issues are not specific to Media Dive but are common issues with ZUIs. No clear answer yet exists as to how these should be addressed. Various suggestions have been made, such as automatic zooming out before zooming in to help users develop an overview of the zoomable space [51, 24, 27], etc.

### 7.2.3. Two Approaches to Constructing Media Dive

Media Dive was implemented in two very different ways as part of the research into evaluating ORRIL. Two different approaches were taken to serve as a contrastive analysis of methods for constructing ZUIs. The first version of Media Dive was written in Java using

Piccolo and a range of other libraries. The second version was built with Nutmeg.

From here on the version of Media Dive written in Java is referred to as MD-Java and the version of Media Dive implemented in Nutmeg is referred to as MD-Nutmeg.

### **Developing Media Dive in Java**

MD-Java was built using a number of Java libraries. Piccolo [5], a low-level Java library for implementing visual ZUIs, was used to create a visual display users could interact with. Without Piccolo every aspect of the visual display would have had to be written from scratch. Piccolo provided an object-orientated visual scene graph, which could be programmed to display various image formats, handle inputs and react to user events. Piccolo is a powerful and flexible library that consists of many hierarchically structured classes with multiple inheritance. It is well designed but under-documented, which meant it sometimes took longer than it should have to understand and use some of the more esoteric classes.

One important issue with Piccolo is that it was not designed to be thread aware. This is important because in Media Dive there are two display types that could be simultaneously active, i.e. the visual display and the audio display. A potential issue with having two simultaneous display types is that the audio display could easily be required to continuously play a song while the visual display remains responsive to user actions. Unless this is handled properly the interface could block, i.e. the audio would play but the visual interface would not be responsive. As a consequence, the audio display was implemented as a set of threads that would not obstruct the visual display threads.

For MD-Java, an audio display had to be created that could play a number of songs at the same time without interfering with the visual display. To achieve this, a number of audio libraries were used, e.g. libraries for handling audio streamed from network servers, for decoding audio formats such as Mp3s and Ogg files. Without these audio libraries, implementing the audio display would have been a very time-consuming development task.

Thread handling and aspects of the thread management were carried out using the standard Java libraries. Each display type took care of its own thread management and sub-threads were spawned and killed as needed with some custom written code.

When the audio display and visual display had been implemented and made to coexist there was enough of a framework in place to get on with actually building Media Dive. This

involved coding the location of the images in the visual display and defining which audio sources were associated with which images. After which more code was written that used Piccolo classes and methods to monitor how the user moved the visual viewport. Next code had to be developed that related user actions and with interface updates. This code stopped or started the audio based on the location of the viewport relative to the image. This involved controlling the audio player threads with code that was called by Piccolo's scene manager.

In all, from start to finish, MD-Java took at least three weeks, required understanding and using various Java libraries and was written in a few thousand lines of Java code. Prior to developing MD-Java, I had a number of years experience coding in Java and numerous other computer languages.

### **Building Media Dive with Nutmeg**

The second version of Media Dive, MD-Nutmeg, was implemented with Nutmeg. This involved specifying in ZIML the layout of the images, and defining what user events to capture and where to capture them. Then a very simple JavaScript was written to start and stop the audio sources. The JavaScript was then associated with specific user events.

Laying out images required knowing how to use a ZIML Object to specify where an image appeared – this is an equivalent skill to knowing how to include an image in a HTML web page.

The JavaScript was very simple: it was a function call to alter the ZIML DOM in real time. Editing the ZIML DOM caused the displays to be instantly updated with any changes made.

The main complication with using Nutmeg was capturing user events. User events are captured with Regions. The Regions in MD-Nutmeg had to be placed in front of the images. If an image is zoomed and there is no Region, then no user event is captured, and no script would be activated to update the display. Specifying the Region in ZIML is easy; the complication is that a Region does not appear on the display. So it can sometimes be hard to know precisely what area a user event is being captured in, e.g. is a zoom event getting captured only when it occurs at the centre of an image rather than at the bottom of an image? This means, specifying Regions can be hard because you cannot see them. Fortunately in Nutmeg a debug mode could be activated that showed a simple visible representation of a Region.

Implementing Media Dive in Nutmeg was a fairly repetitive task. It primarily consisted of:

1. Setting up a ZIML audio Object and ZIML image Object.
2. Creating a Region (to capture user actions) in front of the image Object.
3. Writing a little bit of JavaScript (Interface Logic) to stop and start the audio Object.
4. Creating a Relation that bound user actions to starting and stopping the audio Object.

This task was repetitive because when it had been done once it was a simple matter of copying and pasting the ZIML multiple times and editing a small amount of information in each copy, i.e. the image location, audio location, Region location, etc.

From start to finish, MD-Nutmeg took around two hours to implement. It required understanding and using ZIML and writing some extremely simple JavaScript.

#### **7.2.4. Evaluation of the Approaches**

What skill sets and skill levels were required when writing a Java version when compared with writing a Nutmeg version of Media Dive? Was it easier to build with Java or Nutmeg? What levels of knowledge and abilities were required for building Media Dive?

In the previous sections the two build processes were covered – this highlighted the skills and core tasks involved with creating Media Dive. These can be broadly stated as the abilities to:

- Program
- Create and implement the visual interface
- Create and implement the audio display
- Load and manage media content
- Miscellaneous

Each of these tasks and abilities in MD-Java and MD-Nutmeg is specified in Table 7-1.

## 7. Prototyping Applications with Nutmeg

---

Table 7-1. The tasks and associated skills required when implementing Media Dive in Java and with Nutmeg.

<i>Task</i>	<i>MD-Java</i>	<i>MD-Nutmeg</i>
<i>Programming</i>	Object Orientated programming with Java	HTML like markup language (ZIML) with scripting, e.g. JavaScript
<i>Create Visual Interface</i>	Understand and use Piccolo library and API	Place images with ZIML
<i>Create Audio Display</i>	Integrate audio library and API with Piccolo	Place audio sources and script for starting and stopping songs via ZIML
<i>Load &amp; Manage Media Content</i>	Use file handling API & network protocols for accessing remote data	Not Required (Automatically handled by Nutmeg)
<i>Miscellaneous</i>	Develop multiple thread handling to prevent interface blocking	Not Required (Automatically handled by Nutmeg)

There is a large different in the degree of skills needed for implementing MD-Java and MD-Nutmeg. Nutmeg does not require that users need familiarity with object-oriented programming, or Java development, or multithreading, nor does Nutmeg require that users have the skills to be able to implement their own audio player. This means that one of the important criteria (see Section 4.3) of ORRIL has translated to Nutmeg. It is a simple framework in which each component has a clearly defined role. Users do not have to keep track of multiple threads and they do not have to keep track of multiple classes and methods – they can get on with building ZUIs.

Another important criterion mentioned in Section 4.3 was that ORRIL had to be usable, i.e. it could be used to build ZUIs and not serve merely as a descriptive abstract model. The ability

to construct Media Dive with Nutmeg indicates that ORRIL is usable. Not only is ORRIL usable for creating visual ZUIs, but it can also be used to explore the role of audio in ZUIs.

An additional advantage with Nutmeg is speedier development. MD-Nutmeg was built much faster than MD-Java. While part of this is due to familiarity with ZIML, a strong factor is that the amount of work required is very different. Nutmeg provided a rich environment in which ZUIs could be built. With MD-Java, time had to be spent building the ZUI environment before focus could be given to developing the main functionality in Media Dive. Nutmeg can be used to rapidly prototype ZUIs.

What are the disadvantages? Nutmeg gives you less freedom and control over certain things. It is not byte-oriented so certain kinds of interface component transforms cannot be implemented, e.g. turning on and off pixels in an image. Although it is worth noting that this inbuilt limitation was a design decision motivated by the aim of maintaining simplicity. With Piccolo, the user has control over nearly every aspect of the potential UI – but the disadvantage is the need for a lot more development work and overhead.

Early versions of Nutmeg were used by non-programmers within Media Lab Europe to create interfaces for interacting with audio and image collections.

### **7.3. Implementing Different Forms of Zooming**

There are a variety of different kinds of zooming that may be of use in ZUIs. For example, in Nutmeg the default form of zooming is simple geometric zooming, i.e. an increase or decrease the size of a displayed image. Other forms of zooming such as semantic [41] and bounded box zooming, which is explained below, may also be used. A measure of the success or failure of Nutmeg and ORRIL is how easy it is to implement these different forms of zooming?

In Section 6.8 a simple ZUI application, built with Nutmeg, was described that featured an implementation of semantic zooming. In that example the image that was shown changed based on whether the user zoomed towards or away from the image. Semantic zooming enables people to develop overviews of the content in an interface and then interactively drill down into the details of the content. As can be seen from the ZIML in Section 6.8, implementing semantic zooming in Nutmeg is a simple process of changing what image is associated with an Object. The Interface Logic that changes the association is listed in Section 6.7.

Bounded box zooming, a slight variation on semantic zooming, is where an image is only shown depending on where the viewport is. For example (Figure 7-5), if your viewport was 15 units away from the image's location (at position 1) nothing would appear in the viewport. When the viewport is 10 units away from the image's location (at position 2) the image is shown. Finally when the viewport is 5 units away from the image's location (at position 3) the image is no longer shown.

Bounded box zooming can easily be implemented by editing the ZIML application listed in Section 6.8. Only one change needs to be made to the code. This change is a small rewrite of the Interface Logic listed in imageFlick.js (see Section 6.7). The change is such that when imageFlick.js is run it should:

1. check whether the image "a6" is displayed
2. if it is displayed, turn off the image
3. if it is not displayed, turn on the image

The effect of this change will be that when an *enter* event (<event id="e1" type="enter">) occurs in relation "or1", the image appears, and when the *leave* event (<event id="e2" type="leave">) occurs, the image disappears.

In this section it has been shown that implementing the different forms of zooming (Geometric, Semantic and Bounded) in Nutmeg is a simple process, which reflects well on the underlying design of ORRIL.

Should the different kinds of zooming have been supported as a core property/component of ORRIL? At the moment in Nutmeg the different styles of zooming must be re-implemented time and time again. A justification for not including zoom styles as a core component in ORRIL is that it would introduce another layer of abstraction and component types, which would take from the simplicity of ORRIL. Another argument against including zoom styles is that it is left open for user to invent zooming styles, rather than forcing them to pick from a pre-defined set of styles.

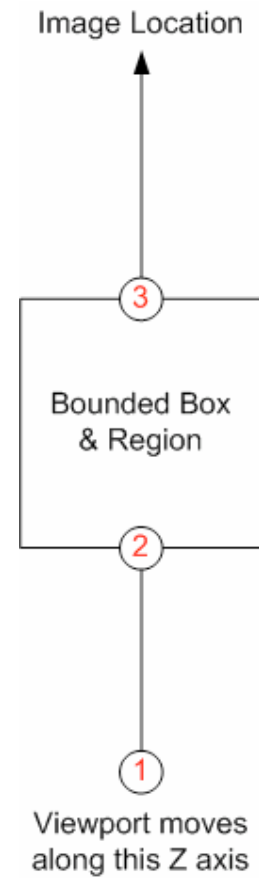


Figure 7-5. Image is shown when viewport is in bounded box.

## **7.4. Conclusions**

This chapter presented an evaluation of using Nutmeg, and indirectly using ORRIL, to create a multi-modal ZUI. Examples of the use of Nutmeg and ZIML for implementing semantic, geometric and bounded box zooming were also presented.

It was shown that Nutmeg could be successfully used for rapidly creating an interactive application, i.e. Media Dive.

Implementing different forms of zooming in Nutmeg was elaborated upon and shown to be a simple process.

Nutmeg was shown to be usable for creating medium to high fidelity zoomable user interfaces.

# Chapter 8

## 8. Conclusions and Contributions

### 8.1. Conclusions

In the course of this thesis and related work the question “How can the process of creating ZUIs be simplified?” has been closely examined. Multiple perspectives and interdependent steps were taken in answering the question and evaluating the resultant answer. During the course of this work there were five primary steps. These steps correspond to a refinement of and focus on specific aspects of the question. The five steps are listed and discussed on the following pages.

**Step 1:** Analyse what is required for constructing ZUIs.

Before an answer to “How can the process of creating ZUIs be simplified?” could be given a clear understanding of what is involved in creating ZUIs was first needed. Without an understanding of the process of creating ZUIs there was no way of knowing how to simplify the process.

In Chapter 3 the ZUI Requirements were identified, defined and explained. Identifying the Requirements lead to a very specific clarification and definition of the properties and process involved in creating ZUIs. The Requirements were shown occurring in a prototype ZUI application called Media Dive.

This was followed with a first attempt, i.e. the Requirement Groups, at simplifying the Requirements. The Requirement Groups served as a general abstraction that was helpful for thinking about how to simplify ZUI construction but was far too abstract to be useful for implementing ZUIs.

**Step 2:** Create an uncomplicated formalism based on what is learnt from the analysis in Step 1.

Step 2 occurred because the stated aim in the research question was to *simplify* the process of ZUI creation. In Chapter 4, simplification was first achieved by taking the Requirements, which served as abstract ZUI compositional units, and making them more specific while also reducing the number of Requirements.

---

Making the Requirements more specific resulted in the creation of a simpler abstraction. This abstraction was called ORRIL. It was shown how ORRIL was based upon the Requirements and Requirement Groups. The use of ORRIL in textual and diagram forms was demonstrated for describing a range of commonly required ZUI actions – this helped establish whether ORRIL was usable for constructing ZUIs.

**Step 3:** Turn the formalism into a simple and usable framework.

ORRIL was converted from an abstract model into a test implementation. This was done to help establish and evaluate the validity and expressiveness of ORRIL for defining and creating ZUIs.

As was explained in Chapter 6 the process of turning the ORRIL formalism into a simple and usable framework led to a close examination of each of the four ORRIL components. Each component was closely examined as it was converted into the markup language ZIML.

Converting ORRIL into a usable framework was easily done. Directly mapping ORRIL into ZIML was readily achieved with no unnecessary new layers of abstraction required. Certain choices were arbitrary when converting from ORRIL to ZIML but these were limited in nature. It was shown that ZIML and indirectly ORRIL are usable for creating ZUIs.

**Step 4:** Implement the framework as a prototyping tool.

After ORRIL was converted into a markup language the next phase in evaluation was to use ZIML (and, indirectly, ORRIL) to create ZUIs. Before this could be done, a tool was required that could read in the ZIML, parse it and render an interactive ZUI based on the contents of the ZIML. This led to the creation of Nutmeg.

As was outlined in Chapter 5 Nutmeg is an experimental tool for rapidly prototyping medium to high fidelity zoomable user interfaces. Nutmeg helped establish what range and style of ZUIs could be defined and implemented with ORRIL. The focus on the simplicity and usability of ORRIL (via Nutmeg) was emphasized and the motivation for certain design decisions was touched upon.

**Step 5:** Attempt to use the prototyping tool to implement ZUIs and some common ZUI interaction techniques.

---

The final step was focused on evaluating whether ORRIL does simplify the process of creating ZUIs. Evaluation was done by attempting to use Nutmeg and ZIML to implement a ZUI application and range of ZUI interaction techniques. If ORRIL did successfully simplify the process of creating ZUIs then ZIML should be relatively easy to use and understand. Furthermore it should not take long to implement the ZUIs, especially when compared with implementing them in the traditional way, i.e. programming them with Java.

In Chapter 7 I presented an evaluation of using Nutmeg, and indirectly using ORRIL, to create a multi-modal ZUI and range of important ZUI interaction techniques. It was shown that Nutmeg could be successfully used for rapidly creating an interactive application, i.e. Media Dive. Implementing different forms of zooming in Nutmeg was elaborated upon and shown to be a simple process.

In this thesis and related work I attempted to simplify the process of creating ZUIs. The result took the form of a solution that enabled the easier creation of ZUIs. The solution was ORRIL, which was implemented in a concrete manner and used to build ZUIs.

## **8.2. Contributions**

The ORRIL framework is useful to implementers of ZUIs because it can serve as part of the basis for the design of ZUI construction tools and development libraries. The framework was generated by the Requirements analysis that was formalised as ORRIL, implemented as ZIML and evaluated via Nutmeg.

Further work on ORRIL could involve establishing whether ORRIL components should be made smarter, i.e. should ORRIL Objects be made intelligent enough to know how to behave under different kinds of zooming (“semantic stacks”)? As ORRIL currently stands zooming behaviours must be implemented with multiple Objects, Regions, Relations and Interface Logic components. This is beneficial in the sense that it does not limit the number of possible kinds of zooming behaviours, but it is disadvantageous because it requires more work by implementers.

## **8.3. Future Directions**

Many questions remain to be examined before ZUIs can or should be deployed in a wide spread manner. Some questions are interface implementation issues, e.g. How should

---

semantic zooming of widgets be defined for developers to control programmatically? Are there algorithms for automating the transitions between representations in semantic zooming?

Other questions are more interface interaction specific. Parts of these question have been touched upon by various research groups, e.g. How should the user control zooming? How can spatial disorientation be prevented and wayfinding enhanced in zoomable displays [4], i.e. stop people getting lost in ZUIs? Does the concept of zooming apply to other modalities, i.e. is there such a thing as zooming touch? Sound is spread out over time so how should semantic zooming occur on it? Are there techniques for general purpose modality independent semantic zooming? What rates of zoom are appropriate [24, 51]? How should scalable interactions (see Section 1.2) be developed? What are appropriate ways of communicating scalable interactions to the user so they know how to interact with a display? What role do other visualization techniques have when used in conjunction with ZUIs, e.g. fisheye views [18, 3], hyperbolic trees [27], etc.

Of course there are many outstanding questions about prototyping ZUIs. For example is it possible to sketch [28] ZUIs and ZUI interface behaviours? Nutmeg was predominately useful for constructing the output displays of ZUIs. How can implementing control of and interaction with ZUIs be simplified [42]? Should there be more Visual Programming Languages [10, 2] for creating ZUIs? Does a ZUI VPL make sense because of the spatial nature of ZUIs, i.e. drag and drop media around a zoomable space to create a ZUI?

One of the major future research directions in ZUIs should cover full featured ZUI desktops. Even a brief glance at the domain of virtual reality turns up numerous research projects [14, 44] that attempt to replace the WIMP desktop with feature rich three dimensional desktops – this should also be done with ZUIs. Much of the research on ZUIs has focused either on single applications, specific ZUI interaction techniques or particular display methods. Yet without putting many of the techniques and theories together into a cohesive whole it is hard to really get a sense of just how successful, or how bad, a ZUI environment could be.

---

## A. XML Schema Definition for ZIML

In this appendix ZIML's XML Schema Definition (XSD) is specified. An XSD formally defines a markup language. An XSD does not have to be created but XML parsers can be told to valid against an XSD to ensure correctness in how a markup language is used.

Further information on how to read XSD's can be obtained from the World Wide Web consortium (W3C) [54].

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <!--
    Definition of ZIML
  -->

  <xs:element name="ziml">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="object" minOccurs="1"
maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <!--
    Definition of ZIML Object
  -->

  <xs:element name="object">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="info" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<xs:element ref="content" minOccurs="1" maxOccurs="1"/>
<xs:element ref="location" minOccurs="1" maxOccurs="1"/>
</xs:sequence>
<xs:attribute name="id" type="xs:ID" use="required"/>
</xs:complexType>
</xs:element>

<xs:element name="info" type="xs:string"/>

<xs:element name="content">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="uri" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
    <xs:attribute ref="type" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="uri" type="xs:string"/>

<!--
  Definition of ZIML Region
-->
<xs:element name="region">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="shape" minOccurs="1" maxOccurs="1"/>
      <xs:element ref="location" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
  <xs:attribute name="id" type="xs:ID" use="required"/>

```

```
</xs:element>

<xs:element name="shape">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="fulc_x" minOccurs="1" maxOccurs="1"/>
      <xs:element ref="fulc_y" minOccurs="1" maxOccurs="1"/>
      <xs:element ref="fulc_z" minOccurs="1" maxOccurs="1"/>
      <xs:element ref="blrc_x" minOccurs="1" maxOccurs="1"/>
      <xs:element ref="blrc_y" minOccurs="1" maxOccurs="1"/>
      <xs:element ref="blrc_z" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
  <xs:attribute ref="type" use="required"/>
</xs:element>

<xs:element name="fulc_x" type="xs:decimal"/>
<xs:element name="fulc_y" type="xs:decimal"/>
<xs:element name="fulc_z" type="xs:decimal"/>
<xs:element name="blrc_x" type="xs:decimal"/>
<xs:element name="blrc_y" type="xs:decimal"/>
<xs:element name="blrc_z" type="xs:decimal"/>

<!--
  Definition of a ZIML Relation

  NOTE: A Relation does not always need both an Object ID
  and a Region ID.

  The relationship between Objects and Regions is:

  N:M where N, M >= 0
```

```
-->
<xs:element name="relation">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="objects" minOccurs="1" maxOccurs="1"/>
      <xs:element ref="regions" minOccurs="1" maxOccurs="1"/>
      <xs:element ref="event" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:attribute name="id" type="xs:ID" use="required"/>
</xs:element>

<xs:element name="objects">
  <xs:attribute name="ids" type="xs:ID" use="required"/>
</xs:element>

<xs:element name="regions">
  <xs:attribute name="ids" type="xs:ID" use="required"/>
</xs:element>

<xs:element name="event">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="uri" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
  <xs:attribute ref="type" use="required"/>
</xs:element>

<!--
```

```
Elements and Attributes which are used in more
than one place
-->
<xs:element name="location">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="x" minOccurs="1" maxOccurs="1"/>
      <xs:element ref="y" minOccurs="1" maxOccurs="1"/>
      <xs:element ref="z" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="type">
  <xs:attribute name="typetxt" type="xs:string"/>
</xs:complexType>

<xs:element name="x" type="xs:decimal"/>

<xs:element name="y" type="xs:decimal"/>

<xs:element name="z" type="xs:decimal"/>

</xs:schema>
```



## B. Definition of Terms

In this thesis I have introduced a number of terms, or used a particular word in a very specific way. This section lists the terms and words with relevant definitions. Definitions start with bold type and words that are defined within this section are in italics.

### **Components**

The units that make up either *ORRIL* (units are: *Objects, Regions, Relations* and *Interface Logic*), the *Requirements* (units are: R1 to R10) or the *Requirement Groups* (units are: *Display, Interaction, Results* and *Data Groups*).

### **Data Group**

Encoded information about content, processes and interactions that can occur in an interface.

### **Display Group**

A display that can be visual, haptic, auditory and/or any other output mode, or fusion of modes.

### **Interaction Group**

Interactions that can be performed with a system.

### **Interface Logic**

*Transforms* that can occur.

### **Objects**

A basic perceptual unit within a zoomable information space, e.g. an image, a piece of audio or a block of text.

### **Occurrences**

Occurrences are defined as user actions, hardware triggers and interface events. Thus occurrences often signify a change of state that should be conveyed via the interface.

## **ORRIL**

*Objects, Regions, Relations and Interface Logic*

### **R1: Render**

Render a display that a user can perceive and interact with.

### **R2: Place**

Place on the display at least one viewport into a very large three-dimensional information space.

### **R3: Allow**

Allow movement of the viewport in the information space so that it can pan and zoom.

### **R4: Constrain**

Constrain the viewport such that it is impossible to rotate it.

### **R5: Position**

Position data, such as text, images and audio, at specific spatial locations within the information space.

### **R6: Transform**

Transform the data based on the viewport position and other occurrences, e.g. user actions.

### **R7: Create**

Create a mapping between which occurrences trigger what transforms of the data.

### **R8: Define**

Define areas within the information space where the mappings exist, i.e. not all transforms will be relevant to all locations or data.

### **R9: Encode**

Encode the nature of the transforms that occur on data.

### **R10: Enable**

Enable transforms and mapping to be altered.

### **Regions**

Regions denote three-dimensional areas where user actions may be captured, e.g. movement of a viewport, continuous updates of a pointer's position, a key press, etc.

### **Relations**

Define the mappings and associations between *Regions*, *Objects* and *Interface Logic*.

### **Requirements**

Ten basic requirements that are needed for the creation of a feature rich *ZUI*.

### **Requirement Groups**

A broader and more abstract simplification of the *Requirements*. Describes interfaces in terms of the *Display*, *Interaction*, *Results* and *Data Groups*.

### **Results Group**

Results are what happen because of an action.

### **Semantic Zooming**

Variation in information representation as a function of scale.

### **Transforms**

Alteration and updating of the content displayed in user interfaces, e.g. altering a range of pixels, changing a block of text, etc.

### **User Action**

User behaviour's that may be captured and acted upon, e.g. key presses, mouse movements, etc.

### **Viewport**

Window into an information space, it may be point based (audio space) or a two

dimensional (visual display) area.

**ZIML**

Zoomable Interface Markup Language

**ZUI**

Zoomable User Interface



## Bibliography

1. Abrams, M., Phanouriou, C., Batongbacal, A., Williams, S. and Shuster, J. UIML: An Appliance-Independent XML User Interface Language. *Proceedings of the World Wide Web Conference*, 1999.
2. Aiken, A., Chen, J., Stonebraker, M., and Woodruff, A. Tioga-2: A Direct Manipulation Database Visualization Environment. *Proceedings of the 12th International Conference on Data Engineering*, pp. 208-217, 1996.
3. Bartram, L., Ho, A., Dill, J., and Henigman, F. The continuous zoom: a constrained fisheye technique for viewing and navigating large information spaces. *Proceedings of the 8th annual ACM symposium on User interface and software technology*, pp. 207-215, 1995.
4. Baudisch, P., Good, N., Bellotti, V., and Schraedley, P. Keeping Things in Context: A Comparative Evaluation of Focus Plus Context Screens, Overviews, and Zooming. *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 259-266, 2003.
5. Bederson, B. B., Grosjean, J. and Meyer, J. Toolkit Design for Interactive Structured Graphics. *IEEE Transactions on Software Engineering*, Vol. 30, No. 8, pp. 535-546.
6. Bederson, B. B., Meyer, J., and Good, L. Jazz: An Extensible Zoomable User Interface Graphics ToolKit in Java. *Proceedings of USIT 2000, ACM Symposium on User Interface Software and Technology*, CHI Letters 2(2): pp. 171-180.
7. Bederson, B. B., and Hollan, J. D. Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics. *Proceedings of the 7th annual ACM symposium on User interface software and technology*, pp. 17-26, 1994.
8. Berners-Lee, T., Fielding, R., and Masinter, L. Uniform Resource Identifiers (URI): Generic Syntax. <http://www.ietf.org/rfc/rfc2396.txt>, August, 1998.
9. Bolt, R. Spatial Data Management, *Interim Report*, MIT Architecture Machine Group, November, 1977
10. Burnett, M. M., *Visual Language Research Bibliography*,

- <http://web.engr.oregonstate.edu/~burnett/vpl.html>, 2005.
11. Bush, V. As We May Think. *The Atlantic Monthly*. Volume 176, July, pp. 101-108, 1945.
  12. Darken, R. P., and Sibert, J. L. Wayfinding strategies and behaviors in large virtual worlds. *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 142-149, 1996.
  13. Donelson, W. C. Spatial Management of Information. *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*. pp. 203-209, 1978.
  14. Elmqvist, N. *3Dwm: Three-Dimensional User Interfaces Using Fast Constructive Solid Geometry*. Masters Thesis, Chalmers University of Technology, Göteborg, 2001.
  15. Engelbart, D. C., and English, W. K. A Research Center for Augmenting Human Intellect. *AFIPS Proceedings of the Fall Joint Computer Conference*, Vol. 33, pp. 395-410, 1968.
  16. Foley, J., Gibbs, C., and Kovacevic, S. A knowledge-based user interface management system. *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 67-72, 1988.
  17. Fox, D. *Tabula Rasa: A Multi-scale User Interface System*. (1998) Doctoral dissertation, New York University, New York, NY
  18. Furnas, G. W. Generalized fisheye views. *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 16-23, 1986.
  19. Furnas, G. W., and Bederson, B. B. Space-Scale Diagrams: Understanding Multiscale Interfaces. *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 234-241, 1995.
  20. Furnas, G. W., and Zhang, X. MuSE: a multiscale editor. *Proceedings of the 11th annual ACM symposium on User interface software and technology*, pp. 107-116, 1998.
  21. Good, L., and Bederson, B. B. Zoomable User Interfaces as a Medium for Slide Show Presentations. *Information Visualization I(1)*, Palgrave Macmillan, pp. 35-49, 2002.

22. Hong, J. I., and Landay, J. A. SATIN: a toolkit for informal ink-based applications. *Proceedings of the 13th annual ACM symposium on User interface software and technology*, pp. 63-72, 2000.
23. IBM's Bean Scripting Framework, <http://jakarta.apache.org/bsf/>
24. Igarashi, T., and Hinckley, K. Speed-dependent automatic zooming for browsing large documents. *Proceedings of the 13th annual ACM symposium on User interface software and technology*, pp. 139-148, 2000.
25. Java SWING, <http://java.sun.com>, 2005.
26. JavaZoom JLayer, <http://www.javazoom.net>, 2005.
27. Lamping, J., Rao, R., and Pirolli, P. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 401-408, 1995.
28. Landay, J., and Myers, B. A. Interactive Sketching for Early Stages of User Interface Design. *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 43-50, 1995.
29. Lecolinet, E. A molecular architecture for creating advanced GUIs. *Proceedings of the 16th annual ACM symposium on User interface software and technology*, pp. 135-144, 2003.
30. Lieberman, H. A multi-scale, multi-layer, translucent virtual space. *Proceedings of the IEEE Conference on Information Visualisation*, pp. 126, 1997.
31. Linton, M. A., Vlissides, J. M., and Calder, P. R. Composing User Interfaces with InterViews. *IEEE Computer*, Vol. 22, No. 2, pp. 8-22, February, 1989.
32. MacKenzie, I. S. and Buxton, W. Prediction of Pointing and Dragging Times in Graphical User Interfaces. *Interacting with Computers*, Vol. 6, No. 2, pp. 213-227, 1994.
33. Microsoft MFC, <http://www.microsoft.com>
34. Microsoft PowerPoint, <http://office.microsoft.com>

35. Myers, B. A. A Brief History of Human Computer Interaction Technology. *ACM interactions*. Vol. 5, No. 2, March, pp. 44-54, 1998.
36. Myers, B. A. Challenges of HCI design and implementation. *ACM interactions*. Vol. 1, No. 1, January, pp. 73-83, 1994.
37. Myers, B. A. User Interface Software Tools. *ACM Transactions on Computer-Human Interaction*, Vol. 2, No. 1, pp. 64-103, March, 1995.
38. Myers, B. A., Giuse, D., Dannenberg, R., Kosbie, D., Pervin, E., Zanden, B. V., Marchal, P. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *Computer*, Vol. 23, Issue. 11, pp. 71-85, November, 1990.
39. Myers, B. A., McDaniel, R., Miller, R., Ferreny, A., Faulring, A., Kyle, B., Mickish, A., Klimovitski, A. and Doane, P. The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Transactions on Software Engineering*, Vol. 23, No. 6, pp. 347-365, June, 1997.
40. Ousterhout, J. K. Tcl and the Tk Toolkit. *Addison-Wesley*, 1994.
41. Perlin, K., and Fox, D. Pad: An Alternative Approach to the Computer Interfaces. *Proceedings of the 20th annual conference on Computer Graphics and Interactive Techniques*, pp. 57-64, 1993.
42. Perlin, K., and Meyer, J. Nested User Interface Components. *Proceedings of the 12th annual ACM symposium on User interface software and technology*, pp. 11-18, 1999.
43. Pook, S. Interaction and Context in Zoomable User Interfaces. *Ph. D. Thesis*, École Nationale Supérieure des Télécommunications, June, 2001.
44. Smith, D. A., Raab, A., Reed, D. P., and Kay, A. Croquet: A Menagerie of New User Interfaces. *Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing*, pp. 4-11, January, 2004.
45. Smith, D. C., and Harslem, E. et al. The Star User Interface: An Overview. *Proceedings of the 1982 National Computer Conference, AFIPS*, 1982.
46. Sutherland, I. E. Sketchpad: A Man-Machine Graphical Communications System. *Ph.D. Thesis*, MIT, 1963.

47. Szekely, P., Luo, P., and Neches, R. Beyond interface builders: model-based interface tools. *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 383-390, 1993.
48. van Dam, A. Post-WIMP User Interfaces. *Communication of the ACM*. Vol. 40, No. 2, February, pp. 63-67, 1997.
49. Wadlow, T. The Xerox Alto Computer. *Byte*. Issue 9, 1981.
50. Walker, M., Takayama, L., and Landay, J. High-Fidelity or Low-Fidelity, Paper or Computer? Choosing Attributes When Testing Web Prototypes. *Proceedings of the Human Factors and Ergonomics Society*, pp. 661-665, 2002.
51. Wallace, A., Savage, J., and Cockburn, A. Rapid visual flow: how fast is too fast? *Proceedings of the fifth conference on Australasian user interface - Volume 28*, pp. 117-122, 2004.
52. Wolfe, A. Toolkit: Longhorn Ties Platform Apps to Core Operating System. *Queue*. Vol. 2, Issue 6, pp. 16-19, September, 2004.
53. Woodruff, A., Landay, J., and Stonebraker, M. Constant Information Density in Zoomable Interfaces. *Proceedings of the Advanced Visual Interfaces Conference 1998*. Italy, May 1998.
54. World Wide Web consortium (W3C), eXtensible Markup Language (XML), <http://www.w3.org/XML/Schema>
55. World Wide Web consortium (W3C), Document Object Model (DOM), <http://www.w3.org/DOM/>
56. Xerox Palo Alto Research Centre History, <http://www.parc.xerox.com/about/history/default.html>