ASSESSING END-USER PROGRAMMING FOR A
GRAPHICS DEVELOPMENT ENVIRONMENT

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER SCIENCE
UNIVERSITY OF REGINA

By
Lizao Fang
Regina, Saskatchewan
November, 2010

© Copyright 2010: Lizao Fang
NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des theses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclu dans la pagination, il n'y aura aucun contenu manquant.

Canada
Lizao Fang, candidate for the degree of Master of Science in Computer Science, has presented a thesis titled, *Assessing End-User Programming for a Graphics Development Environment*, in an oral examination held on August 24, 2010. The following committee members have found the thesis acceptable in form and content, and that the candidate demonstrated satisfactory knowledge of the subject material.

External Examiner: Dr. Luigi Benedicenti, Faculty of Engineering and Applied Science

Supervisor: Dr. Daryl Hepting, Department of Computer Science

Committee Member: Dr. Sandra Zilles, Department of Computer Science

Committee Member: Dr. Robert Hilderman, Department of Computer Science

Chair of Defense: Dr. Katherine Arbuthnott, Campion College
Abstract

The scope of programming is extended, and made more accessible, to satisfy various end-users. Conventional high-level programming languages, such as C++, require strong programming knowledge.

Visual programming attempts to lower the entry barriers to programming allowing the programmer to construct programs by dragging and dropping pictorial components, instead of writing text-based codes. Even though visual programming may be dominant in certain domains, it has not yet become widespread, perhaps due to various issues including increased abstraction and inefficient use of screen real estate.

End-user programming is another attempt to lower the entry barriers to programming that comprises a wide range of activities, some of which require little or no programming knowledge on the part of the user. Visual programming environments can be considered examples of end-user programming systems that do require some programming knowledge. End-user programming aims to offer systems which satisfy common user requirements and are easy to learn and use.

Quartz Composer is a graphics development environment which uses a visual programming paradigm to enable its users to create a wide variety of animations. Although it is very powerful, with a rich set of programming capabilities for its users, there remain barriers to its full use by end-users. Can the provision of additional end-user programming capabilities for Quartz Composer remove some of those barriers? This thesis presents a prototype end-user programming system designed to explore this question. The system, called QEUP, is based on earlier work with cogito. It provides direct access to samples of Quartz Composer output without requiring any of the manual programming involved in Quartz Composer.

Can the end-user programming provided by QEUP contribute to, or replace, the visual programming in Quartz Composer? In order to assess the impact of QEUP, a user study was conducted with 15 participants that compared the visual programming environment of Quartz Composer against the prototype system, QEUP. Preliminary results indicate there may be benefit to using QEUP when first learning Quartz Composer, or when learning new capabilities within it.
Acknowledgements

I would like to take this opportunity to express my sincere gratitude to those who helped me complete this thesis and my graduate studies.

First of all, I would like to thank my supervisor, Dr. Daryl Hepting. He provided me with the opportunity to study in Canada. Furthermore, he offered invaluable guidance in my research. He is the person who inspires my enthusiasm in the research area of Human Computer Interaction. In addition, I appreciate his support in other areas.

I would like to thank Dr. Sandra Zilles, Dr. Robert Hilderman, and Dr. David Gerhard for their assistance in completing this work.

I also would like to thank Mr. Timothy Maciag who offered me advice both about research and life in Canada.

Finally, I would like to thank my parents for their love, support and understanding.
Post-Defense Acknowledgements

I would like to thank my external examiner, Dr. Luigi Benedicenti, for his insightful comments. Also, I would like to thank the Defense Chair, Dr. Katherine Arbuthnott.
Contents

Abstract i

Acknowledgements ii

Post-Defense Acknowledgements iii

Table of Contents iv

Chapter 1 Introduction 1
   1.1 Motivation and Objectives ........................................ 3
   1.2 Outline of Thesis ................................................ 4

Chapter 2 Related Work 5
   2.1 Communicating with Software ...................................... 5
   2.2 Strategies for Learning Software .................................. 6
   2.3 Classifications of Graphical-Object Modelling Systems .......... 7
   2.4 End-user Programming ............................................. 8
   2.5 Visual Programming Environments ................................ 13
   2.6 Gallery-like Interfaces .......................................... 17
   2.7 XML and Applications ............................................ 20
   2.8 Designing Interactive Software Systems ......................... 22

Chapter 3 End-user Programming System – QEUP 26
   3.1 Analysis .......................................................... 26
   3.2 Design .......................................................... 28
List of Tables

3.1 Problem scenario that introduces Jim, an undergraduate student, who is studying Computer Science. .............................. 29
3.2 Activity scenario based on the problem scenario ................... 32
3.3 Information & Interaction scenario 1 (customization) following the activity scenario. ............................................ 33
3.4 Information & Interaction scenario 2 (exploration) following the activity scenario. ............................................ 34
3.5 Patch and Class Names in eupsample.xml .......................... 45
3.6 Part parameter information in eupsample.xml ..................... 46
3.7 Complete parameter information in eupsample.xml ............... 47
3.8 EPPatch, EPParameter and part of their members ............... 48
4.1 Time to complete task (min) ........................................ 60
4.2 Setting values on sample ........................................... 61
4.3 Setting values and connection/disconnection ........................ 62
List of Figures

1.1 User Modelling in Human-Computer Interaction [20]. 2
2.1 The spectrum of software-related activities [64] 9
2.2 Dataflow visual programming 14
2.3 Alice work environment 15
2.4 Quartz Composer: Editor and Viewer 16
2.5 Pure Data work environment 17
2.6 cogito: Each group (A – F) has a representative element (a – f) which is displayed in the Viewer window. b and f are selected for the next iteration of exploration. The space of all possible alternatives is based on $B \cup F$. 18
2.7 Viewer in cogito 19
2.8 New Space dialog in cogito 19
2.9 3-layer SSW Network [15]: Meta Design level, Design level and Use level. 24
3.1 Editor in Quartz Composer 28
3.2 Viewer in Quartz Composer 30
3.3 Editor: load an .qtz sample from local computer 36
3.4 Editor 37
3.5 Description Configuration (part 1) dialog 38
3.6 Description Configuration (part 2) dialog 38
3.7 Editing Default Value List dialog 39
3.8 Adding Value dialog (part 1): add values from Value list 39
3.9 Adding Value dialog (part 2): add values from Manually Input 40
3.10 Adding Value Dialog (part 3): add values from Patches 40
3.11 Viewer ......................................................... 41
3.12 Viewer: select alternatives from the window. The background of selected alternatives becomes green. ......................................................... 42
3.13 Data Flowchart in QEUP (part 1) ........................................ 43
3.14 Data Flowchart in QEUP (part 2) ........................................ 44
4.1 The programming editor window for Quartz Composer, with the example used throughout this paper. ......................................................... 58
A.1 Ethics approval ..................................................... 81
A.2 Consent form ..................................................... 82
A.3 Pre-task questionnaire ............................................. 83
A.4 Tutorial page 1 .................................................... 84
A.5 Tutorial page 2 .................................................... 85
A.6 Tutorial page 3 .................................................... 86
A.7 Tutorial page 4 .................................................... 87
A.8 Tutorial page 5 .................................................... 88
A.9 Tutorial page 6 .................................................... 89
A.10 Tutorial page 7 ................................................... 90
A.11 Tutorial page 8 ................................................... 91
A.12 Tutorial page 9 ................................................... 92
A.13 Tutorial page 10 .................................................. 93
A.14 Post-task questionnaire page 1 .................................... 94
A.15 Post-task questionnaire page 2 .................................... 95
A.16 Post-task questionnaire page 3 .................................... 96
Chapter 1

Introduction

Today, more and more people complete their tasks with the assistance of computers. For an extended period of time, only people with solid computer knowledge could use computers and benefit from them. Working with computers was difficult for the average user. Software was written by experts for their own purposes, and therefore, it was not accessible to everyone.

Domain experts, such as biologists, chemists, and engineers began to demand software they could use without the assistance of a programmer. Matlab\textsuperscript{1} is one example of an application written for domain experts (mathematicians and engineers) which enabled them to process complex computations.

As the computer industry developed, the barriers to accessibility of both hardware and software diminished. The range of application software has grown extensively.

Given a certain software application, two questions emerge:

1. Is the software application capable of satisfying user requirements?

2. Does the software application provide important functionality for users?

Fischer \cite{fischer2000} illustrated these issues in terms of user knowledge with regard to a high-functionality application. The principles can be extended to any software application. In Figure 1.1, D1 represents well-known and frequently-used functions offered by the system. D2 represents seldom-used, but possibly very useful, functions

\footnote{http://www.mathworks.com/matlabcentral/}
offered by the system. Users might not be satisfied with the system if they cannot find some functions which the system actually offers. D3 represents the functions end-users think the system could offer. D4 represents all functions existing in the system. A part of all functions offered by the system are useful to the users, but the rest are not.

The portion of D3 extending beyond the boundaries of D4 causes frustration among end-users, and the concept of end-user programming was introduced as a response. End-user programming allows end-users, with little or no programming knowledge, to conduct programming. The definition seems to be self-contradictory. The former “programming” in the definition refers to writing textual codes in a conventional high-level programming language, such as C, C++, or Java. The latter “programming” includes all creative activities that direct the behaviour of the system. The activities of the first are a subset of the second. The model of end-user programming allows end-users to create their own functions based on existing functions provided by the system.
1.1 Motivation and Objectives

Visual programming, to some extent, lessens the entry barriers to programming. Visual programming employs pictorial components with which to specify programs. It provides an interactive way for users to drag and drop pictorial components and to connect them. So far, visual programming has been only partially successful in certain domains due to the difficulties which Kahn [32] summarized:

1. pictorial components in visual programming increase abstraction because they are symbolic objects
2. pictorial components waste precious screen real estate
3. visual programming inhibits details
4. visual programming does not scale well

The degree of difficulty and complexity of many programming activities is quite varied and the required learning time differs. Studies of end-user programming focus on relatively low-level programming which can be achieved by typical users.

End-user programming encompasses various activities. For example:

1. Using Perl, a conventional script programming language, end-users develop a small tool with which to diagnose and maintain a network.
2. End-users work on Alice, a visual programming environment, to create 3D animations.
3. MS Excel is widely used in industry in accounting and recording of data.

Software systems which allow end-user programming are defined as “end-user programming systems”. The systems should allow end-users to easily operate within the system, as well as reduce the required training time.

In this thesis, visual programming and end-user programming are studied by using two concrete systems. They are Quartz Composer and QEUP (Quartz End User Programming), an end-user programming for Quartz Composer. Two questions are
considered: can end-user programming of QEUP contribute to visual programming in Quartz Composer? Will QEUP be able to replace Quartz Composer? As a comparison to QEUP, a third system, cogito (upon which QEUP is built), is also employed in this research. Fifteen participants joined the user study. Their performance on Quartz Composer, cogito, and QEUP was evaluated and analyzed in order to gain insight into the aforementioned questions.

1.2 Outline of Thesis

The remainder of this thesis is organized as follows:

Chapter 2 presents related work. End-user programming is illustrated in detail. Visual programming environments are examined by describing the characteristics of visual programming, as well as some specific visual programming environments. Two systems, Design Galleries [42] and cogito [28] with gallery-like interfaces are presented. QEUP employs a gallery-like interface. XML and its applications are considered next, since it plays an important role in the QEUP system as a standard format to save and transfer data. The principles employed in the design of interactive systems are described last.

Chapter 3 illustrates the QEUP system, which is an end-user programming system. QEUP is described from three aspects — analysis, design, and implementation. The design section uses scenarios to introduce the functionality of QEUP. The performance of QEUP can be separated into four phases: processing a sample file and pre-configuration; receiving and processing user actions in the Editor; processing data and generating alternatives; and receiving and processing user actions in the Viewer.

Chapter 4 describes the user study and its analysis. The user study is illustrated according to three aspects: participants, materials and task design, and procedure. The analysis of the user study is divided into two parts: analysis of participant performance on the three systems tested and assessment of the impacts of cogito for Quartz Composer, as well as those of QEUP for Quartz Composer.

Chapter 5 presents the conclusion and future work.
Chapter 2

Related Work

Chapter 2 begins with Section 2.1, in which the methods by which users communicate with software systems are presented. Section 2.2 describes effective strategies for learning software. Section 2.3 presents the classifications of graphical-object modelling systems. Section 2.4 provides a detailed presentation of end-user programming, which includes the classification and definition of end-user programming and some principles for evaluation of end-user programming systems. Section 2.5 describes visual programming, including its characteristics and specific examples of visual programming environments. Two systems with gallery-like interfaces, Design Galleries [42] and cogito, are described in Section 2.6. Section 2.7 describes XML, related standards, and their applications. Section 2.8 describes principles for the design of interactive systems.

2.1 Communicating with Software

Fischer [20] argued there are two types of communication channels, explicit and implicit, between users and the software systems. The explicit communication channel is based on user interfaces, including text-based terminal and graphical user interfaces (GUIs). GUIs have been widely accepted and generally well-liked. User interfaces should naturally describe functions provided by systems, significantly decrease errors, and make systems easy to use [50]. The implicit communication channel is based on knowledge. End-users and systems can be in possession of the same basic knowledge.
End-users know the workflow of systems, and systems know what operations end-users could take. Systems can assist users when they determine that users are having problems and interrupt users when they notice inappropriate user operations.

2.2 Strategies for Learning Software

The stereotypical lecture, in which a teacher talks and students listen, is still employed in post-secondary teaching. In this approach, the teacher is positioned as the one who initiates the teaching-learning process. However, this approach is being replaced by a group of educational approaches wherein students are placed at the centre of focus [41]. They include:

1. Active Learning which creates a learning environment where students apply what they are learning in educational activities and think about what they are doing [5, 44].

2. Collaborative Learning is a learning method in which two or more than two students learn something together [18]. Students promote the learning process by sharing their learning experiences and helping each other.

3. Problem-based Learning is a scenario-based learning method in which a set of carefully constructed “problems” are presented to small groups of students [59]. Students gain knowledge by analyzing the “problems”. The student plays an active role in the learning process.

Active, Collaborative, and Problem-based Learning are not mutually exclusive. For example, a researcher asserted that problem-based Learning is a subset of Collaborative Learning, which in turn is a subset of Active Learning [55]. The post-secondary lecture has begun to incorporate the approaches, so students are cognitively active and become the centre of attention in the learning process.

In order to utilize a software system, users must to put effort into learning the system. Because there are two communication channels [20] between users and computers, users need to expend effort and time to learn the software environment and
to gain the knowledge used in the system. In order to reduce the learning time spent, especially on gaining the knowledge used in the system, example-based systems have been introduced.

Example-based systems incorporate student-centered learning methods aimed at reducing the learning time and promoting the quality of learning. Lieberman [39] was one of the first to argue learning by example is the most effective learning strategy. In example-based systems, users specify programs based on simple examples. The samples offered by the systems help users quickly to construct a mental model, by assisting in visualization [54, 39]. The example-based system provides a seed program, and users are able to explore and enrich the program according to their requirements. It is similar to ordering a sandwich at a fast-food restaurant. The restaurant may have certain sandwiches on the menu from which we choose (equivalent to the seed program in the system), but then we may customize the sandwich by changing the type of bread and cheese, by adding pickles, and so on. Various sandwiches are possible, but they are all made according to the same process, which is straightforward and easy to understand. If we are asked instead to first choose then customize a recipe for bread, the task would be much more difficult. Users are able to construct a mental model and gain knowledge through working on simple examples. Furthermore, the knowledge gained in this way is situated within the relevant context of the examples, so the learning process is more effective and users can easily retrieve the knowledge when they need it [10].

2.3 Classifications of Graphical-Object Modelling Systems

Graphical-object Modelling systems are employed to create graphics objects, including bar charts, geometric shapes, cartoons, etc. According to the classification developed by Kochhar et al. [37], graphical-object modelling systems may be classified as either manual, automated, or augmented based on the relationship between humans and computers.

1. Manual systems require the complete involvement of a human in order to construct a graphics application. Users make all the decisions, having complete
control within the modelling processes, and the systems are passively responsi-
sible for the users' operations. Manual systems are widely used in various
domains. MS Paint\(^1\) is a simple graphic design system. Users are provided with
low-level tools, such as pencil and brush, to create graphs. Quartz Composer
is a manual system, as well. Users can drag and drop 2D pictorial elements
to construct programs, which are processed to render graphical objects. They
have to track all parameters to adjust the colour effects and movements of the
graphical objects.

2. Automated systems require no human involvement. Human operations become
unnecessary, even impossible [37]. Computers can automatically produce graph-
ical objects. APT (A Presentation Tool), proposed by Mackinlay [40], automat-
ically achieves graphical presentations of relational information. MS Excel is
able to automatically create various charts based on the data in the spreadsheet.

3. Augmented systems support some notion of the development process as a col-
laborative effort between a human and a computer. Graphic-object modelling
is an ongoing process and is incomplete with users being required to become in-
volved in the modelling process and take some control [29]. Design Galleries [42]
and cogito [28] are two augmented systems allowing users to specify values for
parameters in order to create graphical objects.

2.4 End-user Programming

Ye and Fischer [64] proposed a continuous spectrum of software-related activities.
As illustrated in Figure 2.1, this spectrum includes seven key groups of end-users.
Software professionals are at the right end of the spectrum and they use many
software tools and applications, with an Integrated Development Environment (IDE)
likely the first among them. However, some researchers [64, 16] have pointed out
software professionals may not even use them. Mainstream IDEs in industry include

\(^1\)http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-
us/mspaint_overview.mspx?mfr=true
Visual Studio, Xcode, and Eclipse. SVN and CVS are employed to facilitate collaboration in software development. Other third-party software applications may also be involved in software development. Pure end-users, at the left end of the spectrum, are defined as end-users who passively use software programs to achieve their tasks [16], but they are rare.

Today, almost all software-related activities include some mix of using software and developing software. However, for any given software system, the proportions of using software and developing software may be different. With the exception of pure end-users and software professionals, there are five other groups listed on the spectrum. Costabile [16] describes the five groups in detail.

1. **End-users who customize** perform simple tasks, setting values to parameters. For instance, the Control Panel in Windows and System Preferences in Mac OS X are tools that allow end-users to configure settings for the operating systems.

2. **End-users who write macros** complete their tasks on a relatively simple level. For example, spreadsheet software systems, such as MS Excel and OpenOffice.org Calc, allow end-users to use and adapt macros to automatically calculate and process their data.

3. **Web content developers** are able to set up websites and create web pages, but they have limited knowledge and experience concerning web development. End-users from this group may have received the necessary short term training on web development, but this training time is not enough to assist them in becoming professional web developers.

4. **Developers using domain-specific languages** are domain experts in diverse areas except in the area of Computer Science. End-users in this group have some programming knowledge. Matlab, a domain-specific programming language, is widely used within mathematics and engineering.

5. **Data-intensive researchers** are software developers who are able to manage data and develop software programs. End-users in this group expend minimal effort on software quality efforts, including security and maintenance.

End-user programming (EUP) is defined as the activities end-users, with little or no programming knowledge, perform to create functions or programs. EUP is also called end-user modifiability [21, 22], end-user computing [8], and end-user development [19].

The scope of EUP, proposed by numerous researchers, varies. Ye and Fischer [64] point out EUP systems are software applications which are fully used by pure end-users (the group listed on the left end of spectrum in Figure 2.1). Ye and Fischer defined end-user programming as activities performed only by pure end-users. However, Blackwell [4] categorized EUP activities into five categories:

- based on a scripting language, end-users employ script languages to accomplish tasks. Well known scripting languages include Perl and Python.

- performed within a visual programming environment, such as LabVIEW [31] and Quartz Composer.

- using graphical rewrite systems to complete tasks.
• relying on spreadsheet systems, which are used to manage and process data. They are the most widely used EUP systems. For instance, Spreadsheets for images [38], A1 [33], and MS Excel.

• using example-based programming systems.

According to Blackwell’s classification of EUP, it ranges from customizing values to creating web content. EUP also covers a portion of the use of domain-specific languages. The scope of EUP, proposed by Myers et al. [46], basically covers all software-related activities: spreadsheet systems, visual programming environments, web tools, Visual Basic as business process authoring tools, and Java.

A large percentage of studies of end-user programming focus on relatively low-level programming, which can be achieved by typical users. Based on the above diverse definitions, I present a set of requirements for EUP systems, which can be used to evaluate different EUP systems.

1. **EUP systems support creative activities.** The designers of EUP systems cannot envision the results created by end-users. It is similar to a case where children play with building blocks. They might construct various buildings or other objects in different shapes, which might go far beyond the anticipation of the building blocks’ designers. Utilizing graphics processing systems, end-users are able to create images by following their own ideas.

2. **End-users are able to access EUP systems and benefit from them.** EUP systems do not require specific domain knowledge, only enough experience with computers to empower typical users. Some domain-oriented systems require solid domain knowledge, which restricts their applicability. Matlab is widely used in Engineering, Mathematics and Computer Science. However, other computer users may never access Matlab, nor even be familiar with its capabilities. In contrast, MS Excel is widely accepted by users from various domains, though it was mainly developed for business users.

3. **EUP systems are easy to learn and easy to use.** End-users can become impatient and easily frustrated. End-users do not generally have the time nor inclination
to learn difficult or complex software applications. Furthermore, users want easy
to use systems. Following a short training period, which could be in the order of
an hour, end-users are able to recognize and comprehend a large percentage of
the functions provided by EUP systems. Even without training, end-users could
become acquainted with most functions in some systems. In order to make the
systems easy to learn and easy to use, the development of EUP systems incor-
porates knowledge from multiple disciplines, including Human Computer Inter-
action (HCI), Psychology, Software Engineering and so forth. The ultimate goal
is to allow users to interactively communicate with the systems. As mentioned
in Section 2.2, example-based systems significantly shorten the learning period.
EUP systems can be based on examples as well. Example-based EUP systems
allow users to work on existing examples, providing the seed to be evolved by
users.

4. **EUP systems have the capabilities to resist errors and provide immediate feedback.** There is no restrictive programming syntax as with conventional program-
ing activities. The EUP system can render results without crashing, though
the results could be unreasonable. In contrast, if there are syntax errors in
C++ code, a programmer cannot build a program successfully, and error mes-
sages would be the only results, which are not what programmers expect and
which may not help the programmers correct their errors. EUP systems have a
responsibility to provide immediate feedback, rendering results in real-time or
nearly real-time. Time-consuming EUP systems might deter users.

In fact, most end-users are unwitting end-user programmers [16]. They most
likely think their activities are too simple to be defined as programming. With the
development of computer software and the internet, more and more children are
computer-literate. Children perform design activities and, unwittingly, end-user pro-
gramming [52]. Lego Mindstorms [35] allows children to assemble a robot with plastic
parts, electric motors and sensors. Moreover, Lego Mindstorms offers the additional
capability of allowing children to input commands to control the robot. However, chil-
dren do not believe they are programming. Similarly, most end-users do not realize
they are programming even if they are working on creative designs.

2.5 Visual Programming Environments

Visual programming, or graphical programming, employs pictorial components in order to specify programs. Burnett [12] proposed four categories of visual programming:

1. imperative visual programming by demonstration
2. form/spreadsheet based visual programming
3. dataflow visual programming
4. rule-based visual programming

Of the aforementioned categories, dataflow visual programming is the most widely used [12]. In dataflow visual programming, the pictorial components are connected according to the required data flow. Figure 2.2 shows an example of dataflow visual programming. C1, C2, C3, C4, and C5 represent five two-dimensional components, each of which is a subsystem in a separate context. Components are black boxes which have either input ports or output ports, or both input and output ports. The arrow directions of arrows indicate the direction of data flow. In some other systems, the pictorial components are assembled based on flowcharts. Many visual programming environments (VPEs) allow users to drag and drop the components. The most exciting feature of visual programming is not the colourful graphical components, but the ability to develop, test and modify in an interactive way [11]. The barriers to visual programming are lower than conventional programming [45], and typical end-users with some necessary training can manage to create simple programs in a VPE. Visual programming provides an interactive way for users to perform programming. Some systems support real-time, or approximate real-time, computing. Furthermore, some VPEs such as FPL (First Programming Language) [17] and Quartz Composer are well-suited for typical end-users because the systems eliminate syntactic errors [45].
Next, four VPEs are briefly described. They are ConMan [26], Alice [60, 53], Quartz Composer, and Pure Data.

Haeberli [26] introduced the visual programming environment called ConMan in 1988. ConMan (short for “connection manager”) allowed a programmer to connect simple components to produce a result: each component had input and output ports that could be connected. This is the same paradigm used by Quartz Composer as shown in Figure 2.4.

Alice, shown in Figure 2.3, is an interactive visual programming environment where users can create three-dimensional animations. Alice is used as a precursor to learning an object-oriented programming language, such as Java or C++. Users drag and drop two-dimensional graphical elements in order to create programs. A graphical element could be used to represent an action or a set of actions. Users also need to customize parameters and properties. The system executes programs according to the order of the graphical elements. Alice also provides program flow graphical components, such as If-Then-Else, Loop, and For.

Quartz Composer is another visual programming environment (Figure 2.4). Working on Quartz Composer, a user can create complex graphics and animations including
both two-dimensional and three-dimensional elements. Quartz Composer offers two main windows: the “Editor” within which a developer can specify programs and the “Viewer” which displays real-time results. Figure 2.4 shows an example having 4 components. Components are called patches in Quartz Composer. Patches with different colours represent different types of components. In this example, the Billboard (Figure 2.4, circle 5), in the pink colour, is a Render patch. Render patches only have input ports. All other types of patches are in a green colour. Image Importer, which is used to load images, is connected to the Image port in the Billboard. Two Random number generators (Figure 2.4, circles 2 and 4) and an LFO [Low Frequency Oscillator] (Figure 2.4, circle 3) are used to output data and are connected to the
input ports of Billboard. By clicking Patch Parameters or Patch Inspector, users can specify values for the parameters.

Pure Data (PD) is a visual programming environment, which is widely used for audio, video, and graphics processing. PD is a dataflow visual programming environment and provides immediate feedback to end-users. Figure 2.5 shows the working environment of PD. Similar to Quartz Composer, in order to construct a program in PD, users need to employ patches and draw lines to connect the patches. Both Quartz Composer and PD provide APIs (Application Programming Interfaces) so developers can extend functionality. In order to do this with Quartz Composer, developers must use Objective-C, a text-based programming language. As for PD, developers are required to employ a text-based programming language, such as C or C++. In this context, PD acts as a framework or Software Development Kit (SDK) rather than a visual programming environment.
2.6 Gallery-like Interfaces

Design Galleries [42] and cogito [28] are two systems that allow users to specify values for parameters, which determine the final output alternatives. Systems can generate multiple alternatives each time. All alternatives, which could be images or animations, are shown in a gallery which is comprised of cells. Each cell contains an alternative. End-users recognize, evaluate and select appealing results from this gallery. Because they are straightforward and easily explored, Design Galleries and cogito could be the introductory tools required by high-level creative design.

Design Galleries [42] can automatically process data, generate, and organize results. It also supports multiple selections. Design Galleries has six key elements: input vector, mapping, output vector, dispersion, distance metric, and arrangement.

Design Galleries, cogito and QEUP are able produce and show multiple results each time, so the term “alternative” is used to represent a “result” in these three systems.
Input vector represents a list of input parameters. Dispersion is the phase in which a set of input vectors is processed to create well-distributed output vectors. Mapping is a method to compute data from an input vector to an output vector. Distance metric is used to represent similarity among alternatives.

Hepting [28] proposed cogito to solve some drawbacks of traditional visualization tools. cogito works according to documents generated from other visualization tools. Users customize parameters by selecting values from a list of pre-defined values. cogito automatically generates samples from all possible alternatives and renders them onto multiple cells. Usually, there are eight cells per screen. Users recognize, evaluate, and select appealing alternatives. They can refine alternatives by iteratively customizing parameters. In Figure 2.6, each group (A – F) has a representative element (a – f), which, in turn, has a set of attributes (features). Elements (a – f) are displayed on the cells. If b and f are selected, attributes (features) in group B and F will be combined. The next search iteration is based on $B \cup F$. The space of available alternatives is extended. Cogito solves the issue regarding the limits of human memory, because it instructs computers to work as bookkeepers in order to help users record possible alternatives [28]. Cogito has been used with gnuplot, R, OpenDX and Quartz Composer.
Figure 2.7 Viewer in *cogito*

Figure 2.8 New Space dialog in *cogito*
cogito can be used with Quartz Composer to present the end-user with a variety of outputs, intended to show the breadth of alternatives available by exploring the parameter values of the various patches. Users can iteratively select promising alternatives to establish parameter values for exploration, refine them in the New Space (Figure 2.8) dialogue box, and generate new alternatives for consideration on the Viewer (Figure 2.7) window.

2.7 XML and Applications

XML (eXtensible Markup Language) is a standard for data exchange and representation, recommended by the W3C (World Wide Web Consortium) [63, 62]. Because XML is simple, platform independent and Unicode-based [58], it is widely used for data exchange and data storage.

XML is used as a format for data exchange, particularly on networks [61]. XML makes it possible to transfer and share data across boundaries between two software applications, two networks, two databases, or two operating systems [61]. Using XML systems can easily integrate information from disparate sources, then distribute it to other receivers. WSDL (Web Services Description Language) and SOAP (Simple Object Access Protocol) are two significant elements in Web Service Architecture [23]. Both are XML-based. WSDL is an XML-based module offering descriptions of Web Services. SOAP is a protocol for data exchange. It employs XML as its message format, and relies on HTTP and other Internet protocols. XML is also the specified format for RSS (Really Simple Syndication)\(^5\).

In addition, XML is used as a data storage format [34]. Quartz Composer allows for the saving of documents in Unicode-based XML format. Most software applications have at least one configuration file and the majority are XML documents, although they could have a variety of extension names, such as “.config” under Windows and “.plist” on the Mac. In some situations, saving data in XML format is an alternative to traditional DBMS (Database Management Systems). On the other hand, most DBMS such as Access, MS SQL Server and Oracle, support importing and exporting

data in XML format.

The success of XML is partly due to the accompaniment of other standards. Most standards are defined and recommended by W3C.

XSLT (Extensible Stylesheet Language Transformations) is a standard capable of transforming an XML document into presentation format, such as HTML or PDF, or to another XML document [61]. Bosak asserted that XML and XSLT, together, could be an alternative format for word processing and desktop publishing [6]. It is quite beneficial to transform an XML document to an HTML document. Currently, all mainstream browsers support parsing XSLT and XML. Furthermore, during this transformation, it allows the incorporation of CSS (Cascading Style Sheets).

There are two standards used to validate XML documents, DTD (Document Type Definition) and XML schema. DTD describes the structure of XML documents [51]. It enable users to build well-formed XML documents. The sequence of elements and references can be determined by using DTD. As well, it provides limited data types which are used to restrict element contents and attributes. DTD is a good option when XML documents do not require complex data validation and when there are many XML documents with the same or similar grammar. However, DTD is hard to model complex data, and thus, W3C developed a new validation schema named XML schema, which tends to replace DTD [2]. XML schema has all the functions offered by DTD. Furthermore, XML schema offers further significant functions beyond those provided by DTD. XML schema is an XML-based standard and the processing of XML schema documents does not differ from the processing of common XML documents. XML schema offers various built-in data types, providing a comprehensive and powerful validation system.

In order to use XML, documents in XML must be processed. Based on the classification proposed by Ives et al. [30], there are four main methods to process XML files:

1. **Relational Databases.** Most mainstream DBMSs support the importing and exporting of XML documents to and from databases. XML documents are

---

imported to relational databases. The data can be easily accessed by querying
relational databases instead of parsing the XML document and picking up the
data. However, the method of using relational databases does well only if the
XML documents originate from relational databases, and it could be an ideal
method to process XML documents [30].

2. **XML Databases.** This method solves issues emerging from the importing of
XML documents to relational databases. It preserves the hierarchical structure
of XML documents. Oracle8i [1] is an example of a DBMS that supports XML
databases.

3. **DOM processor.** DOM\(^7\) (Document Object Model) is an interface to access and
update contents in XML documents, as defined and recommended by W3C. To
process an XML document, the system requires the loading of the entire XML
document and creation of a tree-structured document in the memory [27]. DOM
is the optimum choice when the document is accessed repeatedly. The system
need only parse the document once and cache a parsed tree-structured document
in the memory.

4. **SAX processor.** SAX processor is an alternative to the DOM processor. SAX
(Simple API for XML) adopts an event-driven API to process XML documents,
and generates a target document in stream format\(^8\). The SAX processor does not
require as large a memory as the DOM processor. However, the SAX processor
performs less efficiently than the DOM processor, when tasks are involved with
other standards, such as DTD, XLST, and XPath.

### 2.8 Designing Interactive Software Systems

Since the 1970s, diverse users have begun to use computers. Software applications
were developed not only for computing professionals, but also for end-users. "As end-
users became more diverse and less technical, interactive systems came to be compared

\(^7\)http://www.w3.org/DOM/
\(^8\)http://www.saxproject.org/
and evaluated with respect to usability – the quality of a system with respect to ease of learning, ease of use, and user satisfaction. [57]

Presently, the approaches or methodologies of usability engineering are applied to the whole process of software development. They are not just applied to the phases of requirement analysis and software testing. Approaches to usability engineering, executed by HCI experts, can be used to accomplish such tasks as requirement analysis, system design, and usability evaluation.

Scenario-Based Design (SBD) is a systematic method used in usability engineering. The first phase is the analysis phase, including field studies and analysis of different stakeholders in the software system. Based on data collected and analyzed during the analysis phase, designers construct problem domain scenarios. The scenarios are stories which use characters, or actors, based on real users. Problem scenarios are stories that describe stakeholders and their activities in the problem domain. Key features of the problem scenarios are analysed with claims analysis. Relying on problem scenarios and their claim analysis, designers build activity scenarios. Activity scenarios describe interfaces providing users with functionality achieved by the software system. Claim analysis will be executed based on activity scenarios. Information and Interaction scenarios describe how to achieve functionality, as mentioned in the activity scenarios. The two scenarios describe detailed designs. Also, claim analysis is required. The next phase will be lo-fi prototypes, hi-fi prototypes and usability evaluation. Furthermore, in this methodology, in order to make a software application easier and interactive, functionality might be described with the assistance of a metaphor model. Users can easily understand the functionality, thus reducing the burden of learning and recognizing the software system.

Designing an interactive software system involves both software engineering and usability engineering. Work has been done towards consolidating the two endeavours [15, 49, 43].

Costabile [15] proposed a methodology named Software Shaping Workshop (SSW). She described SSW as a three-layer network (see Figure 2.9). This model includes Software Engineers, HCI experts, representatives of end-users and common end-users.
In this methodology, each group, having a different role, works in a separate environment similar to artisan workshops. From the top level to the bottom level, there is a meta design level, a design level, and a use level.

1. **Meta Design level**: W-SE represents workshops for software engineers. Software engineers employ tools and techniques and construct initial programs and are responsible for modifying programs according to requests and comments from the lower level.
2. **Design level**: This level includes workshops for HCI experts (W-HCI) and representatives of end-users (W-RepreW, W-RepreX, W-RepreY, W-RepreZ). The various roles of representatives require different types of workshops. For instance, within the development of an E-commerce business website, there are various stakeholders, including website administrators, registered users, and visitors. The website offers different user interfaces and functionality for the various stakeholders. Representatives bring their domain knowledge and HCI experts bring their knowledge of user interface design, software usability, psychology and human behaviors. They codesign and annotate initial programs brought forward by software engineers.

3. **User level**: This level only includes common end-users. End-users design programs that perform computations for their demands. End-users in this level evaluate programs and communicate with HCI experts and representatives of end-users on the design level in order to help them improve programs.
Chapter 3

End-user Programming System – QEUP

Chapter 3 describes QEUP (Quartz End User Programming), an end-user programming system designed and implemented as part of this research. QEUP is presented from the aspects of analysis, design, and implementation.

3.1 Analysis

The remainder of this thesis will focus on dataflow visual programming, which is currently the most widely used visual programming paradigm [12]. Further references to visual programming will indicate dataflow visual programming.

Many VPEs provide an interactive way to perform programming by dragging and dropping pictorial components. Usually, each component is an isolated subsystem that appears as a black box. Users do not have to understand how the inside of a component works thus, reducing the learning burden. Furthermore, many VPEs eliminate syntactic errors and give immediate feedback to their users.

However, visual programming has not become widespread, apart from some specific domains, for the following reasons:

- visual programming wastes precious screen real estate [32, 9]. Almost all pictorial components are physically larger than the text they replace [45].

- visual programming does not scale well [32]. When the program becomes larger,
users might need to put substantial effort into the on-screen layout of the program. Moreover, visual programming may be harder to comprehend than text-based programming [45, 24].

- visual programming inhibits details. Pictorial representation poorly supports the capture of all details in programs [32, 9].

- pictorial components in visual programming introduce abstraction, because they are symbolic objects [32].

- visual programming lacks portability. Usually, different VPEs employ different visual programming languages. Most visual programming languages require a specific editor, compiler and debugger [45].

- many visual programs are difficult to integrate with programs developed in other programming languages [45]. Because almost all visual programming languages have a specified syntax, it is difficult to integrate the visual programs written in various visual programming languages. Currently, certain visual programming languages are able to interact with other programming languages. Pure Data, for example, allows users to extend the programs using C programming language, and users can enrich the programs in Quartz Composer with the assistance of Objective-C programming language. Generally end-users of VPEs are not professional programmers. End-users could face a nightmare if extending the programs using text-based programming languages.

- many visual programming languages do not support comments [45].

Apart from the above reasons, most visual programming environments require users to have a certain degree of professional domain knowledge. Although the visual programming approach reduces learning by hiding implementation complexities, users could easily become lost during the programming procedure because they lack adequate basic knowledge. In addition, without training, visual programming systems are not easy for beginners. Visual programming requires a period of training time,
Figure 3.1 Editor in Quartz Composer

but end users are “lazy”. Most are not willing to spend substantial time learning a piece of software.

Quartz Composer is a typical visual programming environment, and it shares the drawbacks of visual programming. Using the techniques of Scenario-Based Design, a problem scenario (Table 3.1) can be written to highlight the issues an end-user might face with the Quartz Composer visual programming environment.

3.2 Design

Although the Quartz Composer visual programming environment has many advantages, some shortfalls remain, presenting opportunities for design improvement, namely:
Jim is in his second year in university and he has finished some basic classes in programming and web design. He would like to use Quartz Composer to create a graphic animation. He doesn’t know where to begin so he chooses a template and he is presented with the Editor window shown in Figure 3.1 and the Viewer window shown in Figure 3.2. He finds the 2D box-and-wire representations are appealing, but he is confused about what the boxes represent and he is not sure how to start programming. After some uncertainty, he finds the “Parameters Inspector” button at the top right of Figure 3.1 allowing him to manipulate the parameter values to change the image. However, he is not satisfied because he is not able to connect his manipulations to patches in the editor window. He finds it difficult to make direct changes in the Editor window, and what he can change, he barely understand. He resigns himself to using the parameter dialogue box to make changes. He tries to set values to the parameters by inputing values into the text boxes. He sees some values he sets do affect the results, but others make no visible change. After some manipulations, he realizes what he liked best was something he did a while ago - but there is no history and he cannot recreate exactly what he had before. He is not sure about visual programming.

<table>
<thead>
<tr>
<th>Situation Feature</th>
<th>Possible Pros (+) &amp; Cons(−)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D box-and-wire representations</td>
<td>+ makes the program appealing and the connection relationships are obvious</td>
</tr>
<tr>
<td></td>
<td>+ easy to change the connection relationship between patches</td>
</tr>
<tr>
<td></td>
<td>− increased abstraction makes it hard for learners</td>
</tr>
<tr>
<td></td>
<td>− inhibits details behind the pictorial components</td>
</tr>
<tr>
<td></td>
<td>− wastes the screen real-estate.</td>
</tr>
<tr>
<td>real-time display</td>
<td>+ feedback is immediate, making the programming intuitive</td>
</tr>
<tr>
<td></td>
<td>− is limited to one animation</td>
</tr>
<tr>
<td></td>
<td>− hard to go back to previous results</td>
</tr>
<tr>
<td>limited description (and help information)</td>
<td>+ avoids diverting users’ attention</td>
</tr>
<tr>
<td></td>
<td>− inhibits details making it hard for end-users</td>
</tr>
<tr>
<td></td>
<td>− increases the burden on users’ memories</td>
</tr>
<tr>
<td>text boxes used to set values</td>
<td>+ able to set specific values</td>
</tr>
<tr>
<td></td>
<td>− restricts the scope of exploration</td>
</tr>
<tr>
<td></td>
<td>− makes it difficult to do quick exploration</td>
</tr>
</tbody>
</table>

Table 3.1: Problem scenario that introduces Jim, an undergraduate student, who is studying Computer Science.
Figure 3.2 Viewer in Quartz Composer
• increased abstraction
• details inhibited by pictorial components and limited patch descriptions
• wasted screen real estate
• limited to one animation at a time
• difficult to revisit previous results
• increased burden on users’ memories
• interface restricts the scope of exploration
• difficult to do quick explorations

In SBD, the goal is to reduce or eliminate the negative aspects identified in the problem scenario, while maintaining the positives. Table 3.2 provides an activity scenario, in which the features to address the identified issues are explored.

Based on the problem scenario (Table 3.1), the activity scenario and its claims analysis is described in Table 3.2. The focus in the activity scenario is functionality without concern as to how it will be accomplished. The structure of the interface and the interaction is considered in the Information & Interaction Scenarios, presented in Tables 3.2 and 3.2.

### 3.2.1 Features of QEUP

QEUP is an example-based end-user programming system, which uses examples produced from Quartz Composer. QEUP is intended to be an interactive system which is easy to learn, easy to use, and satisfies user requirements. QEUP should have the ability to support multiple levels of programming activities and to satisfy diverse users with different programming skills. QEUP aims to offer a friendly environment and users are able to accomplish both intentional and subconscious self-study. Working on QEUP, end-users should enrich their cognitive knowledge and prepare for future programming on more complex tasks.

The significant features of QEUP are summarized and listed as follows.
Jim is not comfortable with Quartz Composer. He knows there is a QEUP, which can produce the same results as using Quartz Composer. Jim wants to employ QEUP to create a graphic animation. Because he is a learner of QEUP, he is required to load a basic example when starting the system. Then, in the Editor window, all patches, parameters, and values of the example are listed in a hierarchical structure. He wants to start exploration by manipulating parameter values, but he is asked to customize the description configuration for the patches before exploration. The description configuration includes a brief description of a patch and its parameters, customized names, restrictions for values, and so on. With the assistance of the information collected from the internet and other sources, Jim successfully customizes the description configuration for the patches. He finds the patches belonging to the same class could share the description configuration. During the processes of customizing the description configuration, he realizes he has begun to understand the patches. Then he tries to manipulate the parameter values. He finds he is able to quickly add multiple values to the parameters from lists of values. After setting values for several parameters in which he is interested, he starts building the alternatives. The system processes for a few seconds and multiple alternatives are displayed in a gallery-like window. Jim selects some alternatives which appeal to him. He refines the values and starts the next iteration of exploration. After several iterations of exploration, he finds some alternatives, produced in previous iterations, are better, and he wants to look at them again. Fortunately, he realizes the system saves the history records, and he can easily reload previous alternatives.

<table>
<thead>
<tr>
<th>Situation Feature</th>
<th>Possible Pros (+) &amp; Cons(−)</th>
</tr>
</thead>
<tbody>
<tr>
<td>example-based programming</td>
<td>+ easy to start with</td>
</tr>
<tr>
<td></td>
<td>− limits creativity to some extent</td>
</tr>
<tr>
<td>customizing description</td>
<td>+ accomplishes knowledge accumulation</td>
</tr>
<tr>
<td>configuration</td>
<td>+ eases the burden on users’ memories by recording related description information</td>
</tr>
<tr>
<td></td>
<td>+ eases the burden of programming by predefining restrictions for values</td>
</tr>
<tr>
<td></td>
<td>− takes time and effort to finish description configuration</td>
</tr>
<tr>
<td>adding values from a list of options</td>
<td>+ easy, quick to set values</td>
</tr>
<tr>
<td></td>
<td>− provides limited options for the values</td>
</tr>
<tr>
<td>reloading history records</td>
<td>+ facilitates exploration by supporting rollback</td>
</tr>
<tr>
<td></td>
<td>− not easy to find the right record from a list</td>
</tr>
<tr>
<td>processing multiple alternatives each time</td>
<td>+ improves efficiency</td>
</tr>
<tr>
<td></td>
<td>− does not support real-time display</td>
</tr>
<tr>
<td>selecting alternatives from a gallery-like window</td>
<td>+ easy to use by providing a metaphor that can map to a physical object</td>
</tr>
<tr>
<td></td>
<td>+ could provide many more options</td>
</tr>
<tr>
<td></td>
<td>+ easy to make a decision between alternatives</td>
</tr>
<tr>
<td></td>
<td>− obscures the detail of a certain alternative</td>
</tr>
</tbody>
</table>

Table 3.2: Activity scenario based on the problem scenario
Jim begins to work on QEUP. He finds the dialogue with the title “Select a sample file” (Figure 3.3) and he knows he must load an example. Only those files with .qtz wildcard can be loaded, so Jim selects a .qtz document. In the Editor window (Figure 3.4), he finds 5 patches on the Patches List. They are ImageLoader_1, Billboard_1, LFO_1, Random_1, and Random_2. He knows Random_1 and Random_2 are two patches that belong to the same class. It appears to be a simple example and he thinks he can handle it. Jim decides to work on this example. When Jim clicks Billboard_1 in the Patch List, a message dialogue shows up, saying, “Please define description first if you want to explore.” He knows he must define descriptions before exploration. Jim clicks the “OK” button on the message dialogue to close the dialogue. It leads to the Description Configuration dialogue (Figure 3.5). Under the Node tab, Patch Class and Type have been set with “QCBillboard and “Other”, respectively. Billboard_1 belongs to QCBillboard. Jim completes the descriptions, with the assistance of help information provided by the system.

Next, Jim goes to the panel under the “Parameter” tab (Figure 3.6). On the left, it lists the parameters in the QCBillboard. Jim accesses the sub panel of “inputY” by clicking “inputY”. He sets a readable name for “inputY” on “Defined Name”. Then he sets a proper “Type” and restricts the value range of “inputY” by defining the minimum and maximum values. Then he clicks “Edit” (Figure 3.7) to give names to those values: “0.0” is named “yCenter” and “1.0” is named “Top line”. He clicks “OK”, and the dialogue is closed. Next, he inputs a brief description for “inputY”, saying: “Y axis. Screen coordinates range from -1 to 1. It is “real” type.” Jim continues to customize configurations for the other parameters. After he completes customizing the description configuration, he clicks “OK”.

<table>
<thead>
<tr>
<th>Table 3.3: Information &amp; Interaction scenario 1 (customization) following the activity scenario.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. QEUP is an example-based end-user programming system. When performing programming, based on an example, users could easily begin with this system. The samples offered by the systems help users quickly construct a mental model. Hence, QEUP can significantly reduce users’ learning time.</td>
</tr>
<tr>
<td>2. QEUP could help users achieve knowledge accumulation. According to Human Processor Model [13], when a specific individual tries to finish a task, the responses they selected are mainly determined by knowledge in their Long-Term Memory, because other factors are relatively constant. The richer the related knowledge he/she has, the more reasonable responses he/she could choose. QEUP follows a bottom-up learning method: users start with basic examples, which help them accomplish knowledge accumulation. Based on the knowledge with which they have been acquainted, users are able to manage more complex examples.</td>
</tr>
</tbody>
</table>
Jim begins to work on QEUP. He finds the dialogue with the title “Select a sample file” (Figure 3.3) and he knows he must load an example. Jim selects the .qtz document that he has previously customized (Table 3.2). In the Editor window (Figure 3.4), he finds 5 patches on the Patches List and he notices that a green checkmark appears beside Billboard_1, indicating it can now be explored. Jim clicks “Billboard_1” on the Patches List and all parameters of Billboard_1 are listed on the Parameters List. He selects “inputY”, then its value “Random_1” appears on the Values List. He decides to add other values. Then he clicks the “Add” button and “Add Values” dialogue appears. He finds pre-defined values on the Description Configuration are listed under the “Value List” tab. As well, Jim can add specific values under the “Manually Input” tab (Figure 3.9), and available patches under “Patches” tab (Figure 3.10). He adds several values to “inputY” and clicks “OK”. The Values List on the Editor window lists all values that he just added. Following the same process, Jim adds values to other parameters.

Now, Jim wants to explore the results. He clicks the “Build” button at the top of the Editor window to build alternatives. The system successfully processes the data and the alternatives are generated in seconds. The Viewer window with 8 cells appears (Figure 3.11), and each cell contains selected alternatives. He navigates to “File”, then “Save...”.

Selected alternatives are transferred back to Editor window, which appears on the top level of the screen. Jim refines parameter values by adding new, or removing values. Jim builds a new group of alternatives by clicking the “Build” button, then he picks up attractive alternatives on Viewer window again. He repeats the processes until he is satisfied with the alternatives. After several iterations, Jim wants to retrieve the alternatives produced in a previous iteration. He does this by choosing a record on the “History” List and clicking the “Load” button. Jim wants to save the selected alternatives. He navigates to “File”, then “Save...”.

| Table 3.4: Information & Interaction scenario 2 (exploration) following the activity scenario. |

QEUP starts with .qtz sample documents, which are generated in Quartz Composer. Each sample contains patches, each of which contain multiple parameters. The values of the parameters affect the final results. End-users start with a simple sample, and perform programming based on this sample. End-users are asked to complete necessary descriptions of those patches which are new to them. Meanwhile, users are able to look through descriptions of patches previously defined. Patches are allowed to be explored only if their descriptions have been defined. The processes of defining patch descriptions help users accumulate knowledge. Knowledge accumulation allows users to handle samples at higher levels so they are able to construct more complicated programs.

\[1\] In QEUP and cogito, the term “exploration” is defined as the processes of refining parameters’ values, by either adding or moving values.
3. The Viewer interface of cogito is incorporated as a part of the QEUP system. As noted in Figure 3.11, QEUP inherits the advantages of cogito. As with cogito, QEUP provides an interactive method. It displays alternatives in a gallery of 8 cells per screen. End-users recognize, evaluate, and choose appealing alternatives. The selected alternatives are the foundation for further exploration.

4. End-users specify programs as well as use the programs. For example, end-users are allowed to define types of parameters. If the type is “real” or “integer”, end-users are able to continue defining maximum and minimum values which construct a value range. During the exploration, if values of parameters are out of the defined ranges, the system will pop up a warning message informing end-users of the conflict. Also, users are allowed to edit predefined value lists, which facilitate the selection of values during exploration.

### 3.3 Implementation

Section 3.3 introduces the implementation of QEUP. QEUP is developed on the Mac platform, but designed to be cross-platform. Quartz Composer is a Mac application, but the QEUP system could work with other applications on other platforms. In order to implement the system and make it extendable, I employed C++ as the programming language within Xcode, which is an integrated development environment (IDE) on the Mac. In addition, other software development kits (SDKs) and application programming interfaces (APIs) are used during implementation:

1. wxWidgets\(^2\) is used to develop the GUI. wxWidgets is an open source project. The most important feature of wxWidgets is it supports cross platform development, meaning that it is not necessary to develop different versions of an application for various operating systems. This benefits cross platform software development by shortening the development life cycle, significantly.

\(^2\)http://www.wxwidgets.org/
2. Xerces\(^3\) is another open source project. It offers many APIs for processing XML files. Xerces supports W3C standards related to XML processing. Xerces has two versions: C++ and Java.

3. Core Foundation API is a set of fundamental functions within Mac OS X. Some functions are used to process .qtz documents, which are converted from the format of the binary property list to XML, which then becomes readable. In the XML documents, data is saved in directories, which are pairs of "key" and "value". The value of "key" is saved as a child text node of \textless key\textgreater, and the value of "value" is saved as a child text node of one of following 9 tags. All 9

\(^3\)http://xerces.apache.org/
tags represent value types. They are (string), (real), (true), (false), (date), (data), (array), and (dict). A sample is “inputX” and it has an attribute named “value”, which, in turn, is equal to “0.5”:

```
<key> inputX </key>
<dict>
  <key> value </key>
  <integer> 0.5 </integer>
</dict>
```

Next, we will describe the flowchart of the QEUP system (see Figure 3.13 and Figure 3.14). Circles A and B connect the two flowcharts.
Patch Class: IQCBillboard
Defined Name: !Quad Render 1
Type: Other
Patches: Billboard_1
Description:
It renders a quad positioned with only 2 coordinates.
Billboard also has several unique features:
- Both dimensions (x, y) of the quad can be set directly

Figure 3.5: Description Configuration (part 1) dialog

Figure 3.6: Description Configuration (part 2) dialog
Figure 3.7 Editing Default Value List dialog

Figure 3.8 Adding Value dialog (part 1) add values from Value list
Figure 3.9: Adding Value dialog (part 2): add values from Manually Input

Figure 3.10: Adding Value Dialog (part 3): add values from Patches
3.3.1 Process Sample File and Pre-configuration

A sample document (e.g. eup_sample.qtz) is loaded. The sample document is successfully converted to an XML document (eup_sample.xml) using functions of the Core Foundation API. The XML document can be separated into several snippets. Patch Snippets include patches' names and class names. For example,

```xml
<key>class</key>
<string>QCImageLoader</string>
<key>key</key>
<string>ImageLoader_1</string>
```
According to this code snippet, we can choose information regarding a patch (ImageLoader_1) and its class (QCIImageLoader). Following the same processes, all patches and their classes are successfully chosen (See Table 3 5)

2. Parameter Snippets include parameters’ names, values, and value types. Below is a sample of a Parameter Snippet. Table 3 6 shows the information collected from all Parameter Snippets. In this table, we found the table was missing some other parameters such as “inputX”, and “inputY”. The parameters act as destination ports, accepting data from other external patches. Information regarding such parameters will be picked up from the Connection Snippets.
Figure 3.13: Data Flowchart in QEUP (part 1)
Figure 3.14: Data Flowchart in QEUP (part 2)
Table 3.5: Patch and Class Names in `eupsample.xml`

<table>
<thead>
<tr>
<th>Patch Name</th>
<th>Patch Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageLoader_1</td>
<td>QCImageLoader</td>
</tr>
<tr>
<td>Billboard_1</td>
<td>QCBillboard</td>
</tr>
<tr>
<td>Random_1</td>
<td>QCRandom</td>
</tr>
<tr>
<td>Random_1</td>
<td>QCRandom</td>
</tr>
</tbody>
</table>

3. Connection Snippets store connection information. Every Connection Snippet contains the names of two connected patches. One patch is the source patch, offering data from its output port, and the other patch is the destination patch accepting data from its input port. Also, the snippet contains the type of source
<table>
<thead>
<tr>
<th>Patch Name</th>
<th>Parameter Name</th>
<th>Value</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Billboard_1</td>
<td>inputBlending</td>
<td>1</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>inputColor</td>
<td>1,1,1,1</td>
<td>multiple, real</td>
</tr>
<tr>
<td></td>
<td>inputPixelAligned</td>
<td>false</td>
<td>bool</td>
</tr>
<tr>
<td></td>
<td>inputScale</td>
<td>1</td>
<td>real</td>
</tr>
<tr>
<td>LFO_1</td>
<td>inputAmplitude</td>
<td>1</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>inputOffset</td>
<td>0.0</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>inputPWMRatio</td>
<td>0.20000000000000001</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>inputPeriod</td>
<td>1</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>inputPhase</td>
<td>0.0</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>inputType</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>ImageLoader_1</td>
<td>imageData</td>
<td>defaultImage</td>
<td>external_image</td>
</tr>
<tr>
<td>Randon_1</td>
<td>inputMax</td>
<td>1</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>inputMin</td>
<td>-1</td>
<td>real</td>
</tr>
<tr>
<td>Randon_2</td>
<td>inputMax</td>
<td>1</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>inputMin</td>
<td>0.0</td>
<td>real</td>
</tr>
</tbody>
</table>

Table 3.6: Part parameter information in `eupsample.xml`

patch and the parameter name of the input port. Information collected from the Connection Snippets constructed Table 3.7 and complemented Table 3.6.

```xml
<key>connection_2</key>
<dict>
  <key>destinationNode</key>
  <string>Billboard_1</string>
  <key>destinationPort</key>
  <string>inputX</string>
  <key>sourceNode</key>
  <string>Random_1</string>
  <key>sourcePort</key>
  <string>outputValue</string>
</dict>
```

During the processes of programming, two C++ classes, which are an EPPatch and an EPParameter, are defined (see Table 3.8). Several specific objects of EP-Patch and EPParameter are used to save information in Table 3.6 and Table 3.7.
<table>
<thead>
<tr>
<th>Patch Name</th>
<th>Parameter Name</th>
<th>Value</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Billboard_1</td>
<td>inputBlending</td>
<td>1</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>inputColor</td>
<td>1,1,1,1</td>
<td>multiple, real</td>
</tr>
<tr>
<td></td>
<td>inputPixelAligned</td>
<td>false</td>
<td>bool</td>
</tr>
<tr>
<td></td>
<td>inputScale</td>
<td>1</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>inputY</td>
<td>LFO_1</td>
<td>patch</td>
</tr>
<tr>
<td></td>
<td>inputX</td>
<td>Random_1</td>
<td>patch</td>
</tr>
<tr>
<td></td>
<td>inputImage</td>
<td>ImageLoader_1</td>
<td>patch</td>
</tr>
<tr>
<td></td>
<td>inputRotation</td>
<td>Random_2</td>
<td>patch</td>
</tr>
<tr>
<td>LFO_1</td>
<td>inputAmplitude</td>
<td>1</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>inputOffset</td>
<td>0.0</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>inputPWMRatio</td>
<td>0.2000000000000001</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>inputPeriod</td>
<td>1</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>inputPhase</td>
<td>0.0</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>inputType</td>
<td>0</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>ImageLoader_1</td>
<td>imageData</td>
<td>defaultImage</td>
<td>external_image</td>
</tr>
<tr>
<td>Randon_1</td>
<td>inputMax</td>
<td>1</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>inputMin</td>
<td>-1</td>
<td>real</td>
</tr>
<tr>
<td>Randon_2</td>
<td>inputMax</td>
<td>1</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>inputMin</td>
<td>0.0</td>
<td>real</td>
</tr>
</tbody>
</table>

Table 3.7: Complete parameter information in eupsample.xml

The “valueType” in “EPParameter” is used to store the value type of the parameter. The “valueList” in “EPParameter” is a container employed to save all values of the parameter. Initially, when the sample document is successfully loaded, “valueList” would have one value.

In the scenarios mentioned in previous sections, end-users are requested to define descriptions for patch classes and their parameters. Patches, belonging to the same class, share descriptions. Configuration information is stored in an XML description document (eup_description.xml). The following is a code snippet of the description document.
<table>
<thead>
<tr>
<th>C++ class</th>
<th>Members</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPPatch</td>
<td>name</td>
<td>patch name</td>
</tr>
<tr>
<td></td>
<td>class</td>
<td>class name</td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>patch type</td>
</tr>
<tr>
<td></td>
<td>parameterList</td>
<td>save pointers of its parameters</td>
</tr>
<tr>
<td>EPParameter</td>
<td>name</td>
<td>parameter name</td>
</tr>
<tr>
<td></td>
<td>parent</td>
<td>the pointer of the patch it belongs to</td>
</tr>
<tr>
<td></td>
<td>valueList</td>
<td>save all its values.</td>
</tr>
<tr>
<td></td>
<td>valueType</td>
<td>value type</td>
</tr>
</tbody>
</table>

Table 3.8: EPPatch, EPParameter and part of their members

<node description="Patches belonging to this class are used to generate random values between Max value and Min value, which in turn are two parameters in patches I can define. " definedName="Random Generator" name="QCRandom" type="outputValue">
<parameters>
  <inputMax description="This is a max value that Random patches can produce." max="" min="0" name="max value" type="real">
    <value name="0" val="0"/>
    <value name="1" val="1"/>
    <value name="10" val="10"/>
    <value name="100" val="100"/>
    <value name="360" val="360"/>
  </inputMax>
  <inputMin description="This is a min value that Random patches can produce." max="" min="0" name="min value" type="real">
    <value name="-100" val="-100"/>
    <value name="-10" val="-10"/>
    <value name="-1" val="-1"/>
    <value name="0" val="0"/>
</parameters>
3.3.2 Receive and Process Users’ Actions on Editor

After the system successfully processed the sample document and accomplished the necessary pre-configuration, the controls on the Editor window are bound to the corresponding data. The end-user actions on the Editor would trigger subsystems in QEUP.

1. When end-users start customization on the Description Configuration dialogue (see Figure 3.5 and Figure 3.6), the description document (eup_description.xml) is parsed and the data are transferred to the GUI. Also, the description document would be updated when users finish the configuration.

2. Adding/Removing values on Add Values dialogue (Figure 3.8) requires a pre-defined value list provided by the description document (eup_description.xml). “valueList” in “EPParameter” is updated when users finish adding/removing values.

3. Loading history record results when searching and loading a corresponding cog document. Details of cog documents will be described in Section 3.3.3—“process data and generate alternatives”. “valueList” in “EPParameter” will be updated after successfully loading cog documents.

4. Clicking the “Build” button informs the system to process data and start the next phase—“process data and generate alternatives”.

This snippet records descriptions of “QCRandom”. (node) stores information regarding a patch class. Nodes and attributes correspond to text boxes on the Description Configuration dialogue (see Figure 3.5 and Figure 3.6).
3.3.3 Process Data and Generate Alternatives

If the “Build” button is clicked for the first time on “Editor”, the system would have to generate a template document (eup_standarizedSample.xml) based on eup_sample.xml. Then, data affecting the final alternatives can construct a cog document. QEUP picks up data from the cog document, places them into a copy of a template document, and produces an alternative. Processes regarding the production of the template document are summarized and listed as follows:

1. Replace parameters’ values in eup_sample.xml with: “${patch_name}|{parameter_name}”. Value types would be changed according to the configuration customized by users on Description Configuration.

2. In eup_sample.xml, parameters (e.g. “inputX”, “inputY”, etc.) appearing on Connection Snippets do not show on Parameter Snippets. Items belonging to the parameters must be added to Parameter Snippets during the creation of the template document.

3. The template document does not request Connection Snippets which store connection information among patches. Thus, all Connection Snippets are removed from the document.

Below is a template document code snippet built based on eup_sample.xml.

```
<key>inputPixelAligned </key>
<dict>
<key>value </key>
<bool>$Billboard_1 | inputPixelAligned </bool>
</dict>
<key>inputScale </key>
<dict>
    <key>value </key>
    <real>$Billboard_1 | inputScale </real>
</dict>
```
During exploration, if value types of parameters are not modified by users on the Description Configuration and if it is not the first time the “Build” button is clicked, processes with regard to creating the template document would be skipped.

The next step is to create a cog document which records the value list of the parameters. The cog document is an XML document. The data structure of the cog document is simple. The root node is (cogs), which has multiple child nodes (cog). A (cog) is used to save a parameter’s name and all its values in the value list. (cog) has two attributes: “name” and “value”. “name” is used to save the name of a parameter. All parameter values are separated by “|” and saved into “value”. The structure of (cog) is:

\[
\text{<cog name={$patch\_name | parameter\_name}} \text{ value={$value\_1 | value\_2 | ...}} />
\]

And this is a code snippet in eup.cog:

\[
\text{<cog name=$Billboard\_1 | inputX” value=”Random\_1 | 0.2 | -0.5” />}
\]

In the code snippet, “inputX” in “Billboard_1” has 3 values: Random_1, 0.2, -0.5. The first value is a patch name, representing that “inputX” accepts data produced by Random_1.

After creating a cog document, QEUP begins to validate the parameters’ values. Now, the QEUP system is able to validate whether the values meet value types, and whether values meet value ranges, restricted by type and by maximum and minimum values. Two methods can be used to validate XML documents. They are DTD and XML schema. Details regarding DTD and XML schema have been described in Chapter 2. QEUP employs XML schema to accomplish validation. During the processes of validation, two other documents are created. They are eup_validatedCog.xml and eup_validatedCog.xsd.
1. Data requiring validation in `eup.cog` are transferred to `eup_validatedCog.xml`. The root node in `eup_validatedCog.xml` is `<cogs>`, the same as that in `eup.cog`. Pick up `${patch_name}|{parameter_name}` from `<cogs>` in `eup.cog`, and convert it to the format of `{patch_name} – {parameter_name}` used to construct child nodes of `<cogs>` in `eup_validatedCog.xml`. `{patch_name}–{parameter_name}` has one or more than one child node `<value>`. Values saved in the attribute “value” under `<cog>` in `eup.cog` are separated into individual values and construct child text nodes of `<value>`. Those values, represented by patch names, are skipped and are not used to construct child text nodes of `<value>`. A code snippet in `eup.cog` is:

```
<cog name="$Billboard_1 | inputX" value="Random_1 |0.2| -0.5"/>
```

Data in the above code snippet are picked up and construct a code snippet in `eup_validatedCog.xml`:

```
<Billboard_1-inputX>
    <value>0.2</value>
    <value>-0.5</value>
</Billboard_1-inputX>
```

2. `eup_validatedCog.xsd` is an XML schema document in which there are a set of restrictions used to validate data in `eup_validatedCog.xml`. `eup_validatedCog.xsd` is dynamically created based on configuration customized on Description Configuration. A code snippet in `eup_validatedCog.xsd` is:

```
<x:element name="Billboard_1-inputX">
    <x:complexType>
        <x:sequence>
            <x:element maxOccurs="unbounded" minOccurs="1" name="value">
                <x:simpleType>
                    <x:restriction base="x:decimal">
```

52
The above noted code snippet is used to validate the node \( <Billboard1 - inputX> \) and its child nodes. The fourth line shows \( <Billboard1 - inputX> \) having one, or more than one \( <value> \) as its child nodes. Every \( <value> \) node has a text node whose value type is “decimal”, as well as a value range of “0” to “1”.

Then, \( eup\textunderscore validatedCog.xsd \) is applied to validate \( eup\textunderscore validatedCog.xml \), using functions of Xerces. During the process of validation, if validation fails, reasons and other related information will be captured. QEUP will pop up a message dialogue to inform users. End-users have the responsibility to fix any issues. If the data passes validation, QEUP starts producing alternatives.

To construct an alternative, every parameter in the alternative has a value. In the \( eup\textunderscore cog \) document, every parameter has one or more than one value. For each parameter, if it has \( n \) values, the probability an individual value is picked is \( \frac{1}{n} \). To generate an alternative, every parameter picks up a value from its value list. The probability of producing a unique alternative is \( \prod_{i=1}^{m} \frac{1}{n_i} \) (\( m \): the number of parameters; \( n_i \): the number of values in the value list of the \( i \)th parameter). The total number of possible alternatives is \( \prod_{i=1}^{m} n_i \) (\( m \): the number of parameters; \( n_i \): the number of values in the value list of the \( i \)th parameter). QEUP generates all possible combinations, each of which, in the end, will produce an alternative. The combinations are stored in “EPData”, which is a C++ vector.

Based on the template document \( eup\textunderscore standarizedSample.xml \), previously produced, QEUP replaces each “\$\{patch\textunderscore name\}|\{parameter\textunderscore name\}” with a corresponding value in a combination being saved in “EPData”. If parameters’ values are patch
names, there is a connective relationship between the two patches. The connection codes follow the same data structure of Connection Snippets as mentioned in the first phase—“process sample file and pre-configuration”. Following the same processes, QEUP produces other alternatives. If the number of alternatives is less or equal to 8, all possible alternatives will be generated. However, if the number of alternatives is more than 8, 8 combinations are selected from “EPData”. Each time QEUP will produce 8 alternatives following the order in “EPData”.

On “Viewer”, when users navigate to the next screen by clicking the ”Next Screen” arrow, QEUP will process and generate the next 8 alternatives, if they do not exist.

In addition, if data passes validation, the cog document is renamed and saved as a history record.

3.3.4 Receive and Process Users’ Actions on Viewer

Alternatives are displayed on cells on “Viewer”. Users recognize, evaluate, and select alternatives. Responding to users’ operations on “Viewer”, QEUP is triggered and starts to perform corresponding tasks.

1. Clicking the “Editor” button, the “Editor” window is activated and shows on the top level of the screen. No calculation is required.

2. Navigating the next screen by clicking “Next Screen” requests QEUP to generate the next 8 alternatives if they have not been generated previously. Section 3.3.3 mentions that QEUP produces, at most, 8 alternatives each time. If the screen has been visited previously, QEUP need not produce alternatives again.

3. Sorting alternatives by a certain keyword informs QEUP to sort combinations in “EPData”. Quicksort will be applied to sort combinations in “EPData”.

4. Selecting alternatives and continuing to the next iterative exploration leads QEUP to transfer data of selected alternatives to “Editor”. According to the
principles of *cogito*, as shown in Figure 2.6, parameters' values in selected alternatives construct a new data set. The new values for parameters append to the Value List in the Editor window.
Chapter 4

User Study and Analysis

A user study was conducted in order to assess the impact of QEUP for Quartz Composer. Approval for the study was obtained from the Research Ethics Board at the University of Regina (Figure A.1). 15 participants were divided equally between the 3 different conditions described below. The between-subjects study was designed to examine users’ experiences with Quartz Composer in three different cases.

4.1 Participants

Fifteen participants, ranging in age from 18 to 32, took part in the study. All are current students at the University of Regina. Based on their responses to the pre-task questionnaire (Figure A.3), their areas of study are diverse: Science, Business and Arts/Humanities. Most have taken at least one Computer Science course. With regard to their level of programming knowledge, 4 participants reported a low level, 5 reported a medium level, and 6 reported a high level. Fourteen participants reported no experience with Quartz Composer, and 1 participant reported low experience on Quartz Composer. They had at most a medium experience with visual programming. All 15 participants described themselves as mathematically inclined, but only 1 participant described himself/herself as artistically inclined.
4.2 Materials and Tasks Design

The fifteen participants were randomly assigned to 1 of 3 groups. The backgrounds of the individuals in each group do not prove different. Each group used Quartz Composer directly (QC Group). Participants in the second group used cogito, followed by Quartz Composer (cogito Group). Participants in the third group used QEUP, followed by Quartz Composer (QEUP Group).

A between-subjects design [3], which means that a participant was only assigned 1 condition, was employed. The goal was to assess participant performance with Quartz Composer.

Each participant also encountered the following documents:

1. A pre-task questionnaire (see Figure A.3) which covered aspects of the participants’ background, including education and experience with computer programming.

2. A tutorial manual (see Figure A.4) which provided a standard introduction to the software systems being used. Training is a part of the knowledge acquisition phase, and in particular, it is an important phase for beginners.

3. A post-task questionnaire (Figure A.14), which captured aspects of their experience with the software system(s) used by participants. Questions involved aspects of the participants’ cognition, perception, and comprehension of the software system(s), as well as programming problems.

During each segment of the study, participants began with a very simple sample (shown in Figure 4.1). The sample had only 5 patches: an Image Importer (to load the image), a Billboard (to display the image), an LFO (low-frequency oscillator) and 2 Random patches, both used to generate values that moved the Billboard around the screen. Participants were requested to work on the input sample, refine the sample, and produce results.
4.3 Procedure

Participants completed the consent form and a pre-task questionnaire. The QC Group received some training, during which the study instructor helped them look through the QC Group tutorial material (A.5, A.6) in Appendix A, and then used Quartz Composer for a maximum period of 15 minutes. The cogito Group received some training, during which the researcher helped them look through cogito Group tutorial material (A.5, A.6, A.7, A.8, A.9) in Appendix A, and then used cogito, followed by Quartz Composer, each for a maximum period of 15 minutes. The QEUP Group received some training, during which the researcher helped them look through
QEUP Group tutorial material (A.5, A.6, A.10, A.11, A.12, A.13) in Appendix A, and then used QEUP, followed by Quartz Composer, each for a maximum period of 15 minutes. Participants were asked to talk about what they were doing as they navigated the software applications, using a think-aloud protocol. Each participant also completed a post-task questionnaire.

All operations using the software were recorded from the computer screen, as well as audio from the participants’ interactions (the participants themselves were not video recorded).

4.4 Results and Analysis

Through analysis of the video recordings and the post-task questionnaires (shown in Figure A.14), two questions were considered. First, can QEUP affect the users’ behaviours in Quartz Composer? Second, does QEUP have the capacity to replace Quartz Composer? This section first analyses participant performance with Quartz Composer, then summarizes the three software systems in terms of their advantages and disadvantages.

4.4.1 User Performance on Quartz Composer

All data in the following tables was collected from performance on Quartz Composer. The participants’ performances were analyzed from three aspects: time to complete the task, ratio of setting values, and ratio of setting values divided by connection/disconnection. The final outputs in Quartz Composer is determined by parameter values and the connection relationships among patches.

Time to complete the task

In the study, the time spent on Quartz Composer was limited to 15 minutes. Participants were instructed to stop the task before the 15 minutes expired, if they were satisfied with their results. Table 4.1 shows the time all participants from all three groups spent on Quartz Composer. Two participants in the QC Group, one
<table>
<thead>
<tr>
<th>Group</th>
<th>Participant</th>
<th>Times</th>
<th>Total</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>QC</td>
<td></td>
<td>10.00</td>
<td>13.28</td>
<td>15.00</td>
</tr>
<tr>
<td>cogito</td>
<td></td>
<td>14.25</td>
<td>15.00</td>
<td>15.00</td>
</tr>
<tr>
<td>QEUP</td>
<td></td>
<td>9.22</td>
<td>10.25</td>
<td>11.00</td>
</tr>
</tbody>
</table>

Table 4.1: Time to complete task (min)

in the *cogito* Group, and four in the QEUP Group were able to produce satisfying results within the 15 minute limit. The other participants did not indicate they had produced satisfying results within the 15 minute limit, and were asked to stop. The total and average time spent by the QEUP Group is shorter than for the other two groups.

All participants are beginners with Quartz Composer. However, participants who used QEUP seem to produce satisfying outputs more efficiently. Users communicate with computers based on two channels: explicit and implicit channels [20]. The explicit channel is based on the user interface and the implicit channel relies on knowledge. As mentioned above, the participants in this study have similar backgrounds, and they have no, or very limited, experience on Quartz Composer. Hence, their communication with the computer, through the explicit channel, is not very different. The improvement of the communication through the implicit channel may be the factor that results in less time spent by the QEUP Group. The QEUP Group might gain the necessary knowledge of Quartz Composer after they have some experience on the QEUP system.

**Ratio of setting values**

All participants perform programming based on the example. In order to set a suitable value for a parameter, participants might try several times. The ratio of setting values, which is equal to set-value operation attempts divided by set-value operations kept, is calculated as an indicator of how quickly parameter values are set.
The ratio represents how many trial values are needed in order to successfully customize a parameter. The data in Table 4.2 is related to the example only. Operations on new patches were not considered. Table 4.2 shows data from the three groups. The minimum ratio is 1, which reasonably occurs when participants do not set any values.

The QEUP Group has the smallest ratio, which may indicate use of the QEUP system helps participants understand which parameter values they want to use in Quartz Composer, so fewer changes are required. The significance of using cogito on this aspect is less obvious. As well, Table 4.2 supports the assertion previously mentioned that the QEUP Group participants’ knowledge, including cognition, perception, and comprehension, are improved by using QEUP first.

### Ratio of setting values divided by connection/disconnection

Setting values and connection/disconnection are two groups of operations that determined the final outputs. The operations on both new patches and the patches
in the example are considered in this section. As noted in Table 4.3, set-value operations is the number of times values are set, (dis)connection operations is the number of connections/disconnections, and ratio is the number of set-value operations divided by (dis)connection operations. If the ratio is greater than 1, values were set more often. If the ratio is less than 1, (dis)connecting patches was done more often. Following Table 4.3, the QC Group has no participants whose primary operation was connecting/disconnecting patches. However, there are 1 in the cogito Group and 3 in the QEUP Group. Yet, even the QEUP Group contains participants whose primary operation was setting values.

Following the classification developed by Kochhar et al. [37], graphics development environments may be classified as either manual, automated, or augmented. Manual systems require the complete involvement of a human to construct a graphics application; automated systems require no human involvement; and augmented systems support some notion of the development process as a collaborative effort between a human and a computer.

Table 4.3: Setting values and connection/disconnection

<table>
<thead>
<tr>
<th>Group</th>
<th>User Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>QC</td>
<td>Set-value operations</td>
</tr>
<tr>
<td></td>
<td>(Dis)connection operations</td>
</tr>
<tr>
<td></td>
<td>Ratio</td>
</tr>
<tr>
<td>cogito</td>
<td>Set-value operations</td>
</tr>
<tr>
<td></td>
<td>(Dis)connection operations</td>
</tr>
<tr>
<td></td>
<td>Ratio</td>
</tr>
<tr>
<td>QEUP</td>
<td>Set-value operations</td>
</tr>
<tr>
<td></td>
<td>(Dis)connection operations</td>
</tr>
<tr>
<td></td>
<td>Ratio</td>
</tr>
</tbody>
</table>
In the QC Group, participants seem to expend far more effort in tracking parameter values. However, participants in the cogito Group and QEUP Group tend to be less concerned about tracking parameter values. Therefore, Quartz Composer could be defined as a manual system, and the other two systems, cogito and QEUP, could be augmented systems.

The Patch Inspector (Figure 4.1(b)) window for setting values is a window on the second level in Quartz Composer. It is accessible by clicking the Patch Inspector button on the Editor Window. Connection/disconnection is an operation in the Editor Window, which is the first level window. The operations of setting values are on a lower level, though they are important. Using QEUP and cogito systems might inspire participants to move from a lower level to a higher level. In addition, the impacts of using QEUP on this aspect are more noticeable.

4.4.2 Analysis of Three Systems

Norman [48] pointed out there are two gulfs between users and the software running on computers. The gulf of execution represents the difference between users' intentions and available actions provided by the system. The gulf of evaluation describes an effort to evaluate whether the representations offered by the system meet their expectations and intentions. Ko et al. [36] shared the characteristics of the gulfs of execution and evaluation, and proposed six learning barriers in end-user programming systems.

- Design Barriers: users do not know what they want the computer to do.
- Selection Barriers: users know what they want the computer to do but do not know what to use.
- Coordination Barriers: users know what to use but do not know how to combine it with other elements.
- Use Barriers: users know what to use but do not know how to use it.
- Understanding Barriers: users know how to use the system but the output is not what they expect.
• Information Barriers: users know why the system cannot meet their expectations but do not know how to check.

Inspired by the above six learning barriers, based on explicit and implicit communication channels [20] between humans and the computers, we group the issues of users' performances in Quartz Composer, cogito, and QEUP into two learning barriers. Performance issues with regard to the explicit communication channel are defined as Interface Barriers. Those dealing with the implicit communication channel are called Knowledge Barriers. Next, we will present the three systems from the aspects of their Interface Barriers, Knowledge Barriers and Advantages. The summary and conclusion will be presented at the end of this section.

Interface Barriers

1. Interface Barriers in Quartz Composer

**IB1.1** Poor support for undo/redo, both when setting values through the Patch Inspector and when drawing connections between patches.

**IB1.2** Lack of transparency in the way values are changed following connection/disconnection.

**IB1.3** A large number of patches make the programs cluttered. Though Quartz Composer supports the creation of macros in order to group multiple patches into a single macro patch, an abstraction is introduced, which increases users' burden.

**IB1.4** Clickable targets (such as the small circles on patches to directly change parameter values), may be too small for fast and accurate clicking.

2. Interface Barriers in cogito

**IB2.1** As related to the pre-defined value lists in the New Space Dialog, all options in the value lists are presented in text format. With regard to colour parameters, showing samples of the colours would be more appropriate.
Without an end-user programming capability, the items in the value lists cannot be changed.

**IB2.2** The 17 parameters in the sample are presented in a scrollable window without any hierarchical structure. Participants reported they become easily lost tracking the connection between two patches. It is indicated cogito would not scale well to more complex examples.

3. Interface Barriers in QEUP

**IB3.1** The connection relationship among the patches becomes easily lost.

Going back and forth between Editor and Viewer windows is not intuitive nor interactive.

**IB3.2** It does not support real-time display. QEUP and cogito produce multiple alternatives at the expense of real-time display.

**Knowledge Barriers**

1. Knowledge Barriers in Quartz Composer

**KB1.1** Participants might try repeatedly to set a suitable value to a parameter. The number of trial values varies among the different participants and the different groups. The number of trial values may be related to the degree a user understands the parameter.

**KB1.2** A large percentage of participants wanted to add new patches beyond those contained in the sample, but not all succeeded. Few participants paid much attention to the description documents when adding patches. New patches were continually added, based on patch names: if the name was not intuitive or out of their knowledge scope, the patch would have little chance of being added or used.

**KB1.3** Participants became frustrated when they had ideas but didn’t know how to implement them. For instance, one participant spent a considerable amount of time trying to add a gradient background, but still failed. In
terms of the six learning barriers proposed by Ko, Myers and Aung [36], 
this is an instance of the Selection Barrier or the Coordinate Barrier.

2. Knowledge Barriers in cogito

**KB2.1** cogito does not provide a description of the patches. Even though the 
text regarding the value options is descriptive, it may not be easy for a 
user to understand them in context.

**KB2.2** cogito is abstract and confusing for some participants. Relationships  
between patches are not clear, and this significantly increases the burden of  
forming mental models.

3. Knowledge Barriers in QEUP

**KB3.1** QEUP is a little difficult to use in the beginning. This is a new system  
and no other similar systems exist. In contrast, Quartz Composer has  
many similar software applications. It is easier for beginners if they have  
experience with similar software applications. As well, the interfaces of  
QEUP are not as attractive as those in Quartz Composer. But as long as  
users take the time to become familiar with QEUP, the user’s ability to  
produce graphical objects increases significantly.

**Advantages**

1. Quartz Composer provides an interactive method to perform programming. It  
allows users to drag and drop pictorial components. In the post-task ques-
tionnaire, seven participants rated the functionality of dragging and dropping  
pictorial components as their most favourite feature. Seven participants rated  
the real-time display as their favourite feature. Additionally, in Quartz Com-
poser, the clarity of relationships and data flow among the patches is important.

2. Cogito significantly reduces the time expended on each iteration of exploration.  
QEUP asks users to go back and forth between the Viewer and Editor windows.  
However, all operations in cogito are executed on the Viewer window. There is
a tradeoff between efficiency and abstraction. *cogito* improves efficiency, but it increases abstraction. As a consequence, it increases the difficulty of building mental models. When users become familiar with *cogito*, the efficiency could overcome the abstraction. *Cogito* is able to produce and display multiple alternatives each time.

3. QEUP allows users to customize and personalize programs. A participant said it was “like a programming language without syntax” (see Appendix B). The phase to customize Description Configuration assists users in finishing the knowledge acquisition phase, by accomplishing knowledge accumulation. In addition, in the QEUP system, end-users construct programs, and use the programs to explore parameter combinations within Quartz Composer.

**Summary**

The analysis from the Interface Barriers, Knowledge Barriers and Advantages of three systems, shows:

1. researchers [45, 32, 9] (see Section 2.5) have identified several reasons why visual programming has not become widespread. The Interface Barriers and Knowledge Barriers emerging in Quartz Composer have validated some of the reasons.

   • pictorial components in Quartz Composer increase abstraction, because they are symbolic objects (IB1.2).

   • pictorial components waste precious screen real estate (IB1.3, KB1.1).

   • Quartz Composer inhibits details (IB1.2, KB1.1, KB1.2, KB1.3).

   • Quartz Composer does not scale well (IB1.3).

System designers are able to lower barriers caused by the native characteristics of visual programming. But, they cannot remove the barriers since they inevitably exist in Quartz Composer. For example, pictorial components in Quartz Composer waste precious screen real estate. System designers can reduce this barrier by organizing the components or decreasing the size of the components (but this
may make the clickable targets more difficult to hit quickly). Unfortunately, this barrier is hard to remove. In order to save the screen real estate, one could employ the text instead of the pictorial components, but the system might not remain a visual programming environment.

2. As with Quartz Composer, some barriers emerging in cogito and QEUP are inevitable. However, others are not. There is a tradeoff between displaying an output in real-time and displaying multiple outputs each time. Quartz Composer supports real-time display but does not support multiple display. cogito and QEUP can process and display multiple outputs each time, but they cannot display the output in real-time. Some participants in the study rated the lack of real-time display as their least favourite feature in QEUP. This barrier can be reduced by improving processing speed, but it cannot be removed. Some other barriers in cogito and QEUP are caused by inappropriate designs and these barriers can be overcome by “good” designs. For example, the parameter list in New Space dialogue in cogito is not well organized. The connection relationships among patches can become easily lost.

Generally speaking, QEUP inherits the advantages of the cogito and significantly reduces or removes the barriers in cogito and Quartz Composer. QEUP is an augmented system. It is “like a programming language without syntax”. QEUP allows ordinary end-users to specify programs.

3. In order to improve the design of QEUP, it is possible to incorporate more of the benefits of visual programming. For example, the connection relationships among patches are easily lost in QEUP, but they are clear in Quartz Composer. QEUP could introduce pictorial components to represent the connection relationships. Furthermore, it could be useful and efficient to the point users are able to define the connection relationships directly on the components in an interactive way. However, the component supporting visual programming is an auxiliary part and will not become a core part in QEUP. So far, many software systems have accepted visual programming as its auxiliary component. In MS Access, end-users define foreign keys by drawing lines to connect related tables.
In addition, the relationship among tables is obvious. Some IDEs, such as Visual Studio and Xcode, support visual programming, allowing programmers to drag and drop pictorial components in order to design user interfaces.
Chapter 5

Conclusion and Future Work

Quartz Composer is a visual programming environment, as well as a graphics development environment. Quartz Composer provides an interactive method for end-users to perform programming. End-users drag and drop patches, then connect them by drawing lines from source ports to destination ports. As with other visual programming environments, Quartz Composer has several drawbacks. Pictorial components in Quartz Composer increase abstraction, as well as waste precious screen real estate. Also, Quartz Composer inhibits details and does not scale well. Furthermore, Quartz Composer is hard for learners, because it does not provide enough description information, and does not follow the bottom-up recursive learning method.

Another programming model named “end-user programming”, is proposed in order to overcome the barriers emerging in Quartz Composer. End-user programming (EUP) is defined as the activities that end-users, with little or no programming knowledge, perform to create functions or programs. We are able to evaluate EUP systems from four aspects. Firstly, EUP systems support creative activities. Secondly, end-users are able to access EUP systems and benefit from them. Thirdly, EUP systems are easy to learn and easy to use. Finally, EUP systems have the capabilities to resist errors and provide immediate feedback.

QEUP is an example-based end-user programming system. Users start with a basic example, following the bottom-up recursive learning method. Specific examples aid visualization [39] and assist users to create concrete mental models. Description information is highlighted in QEUP. QEUP requires end-users to define a necessary
description of the patches and their parameters. On one hand, it helps end-users accomplish knowledge accumulation. On the other hand, it allows them to customize description configuration and they can access the information anytime. As a consequence, it reduces the burden on users’ memories. Unlike Quartz Composer, which is a manual system, QEUP is an augmented system. In Quartz Composer, users must track the value of each parameter in order to identify its effect on the final output. However, in QEUP, users are able to set multiple values for each parameter, and the system processes and generates many outputs each time. Users’ attention is shifted from setting values to making decisions as to selecting outputs from diverse alternatives.

Can QEUP affect users’ behaviours on Quartz Composer? Does QEUP have the capacity to replace Quartz Composer? According to the analysis of the user study, QEUP is useful as a complementary system for users who perform programming on Quartz Composer. QEUP helps end-users achieve knowledge acquisition and form mental models. Participants starting with QEUP, followed by Quartz Composer, are better able to cope with the barriers that emerge with Quartz Composer. It may be, experience with QEUP can provide a headstart in learning the Quartz Composer graphical development environment. Using QEUP helps users make better decisions, and shifts their focuses from a lower level to a higher level. Furthermore, it provides a similar benefit when a user wishes to better understand new features and new patches within Quartz Composer.

As a visual programming environment, Quartz Composer provides an interactive method for users to specify programs, such as displaying the output in real-time and dragging and dropping pictorial components. However, there are some barriers in Quartz Composer, both Interface Barriers and Knowledge Barriers. For example, Quartz Composer increases abstraction, does not follow bottom-up or top-down learning methods, and does not offer necessary description. QEUP is an example-based end-user programming system. It is an augmented system, shifting users’ attention from setting values to making decisions regarding the selection of outputs from diverse alternatives. In addition, QEUP supports creative activities, eases the burden on memory, and helps users accomplish knowledge accumulation. QEUP could be a
potential alternative to Quartz Composer, although, at present, it is not a mature system and requires improvements.

Software systems can incorporate the technology of visual programming as its auxiliary components. QEUP could employ visual programming to represent the connection relationship among patches and allow users to modify the connection. Many software systems have incorporated the technology of visual programming to facilitate users interface designs and relationship setting. Visual Studio and Xcode support visual programming to design user interfaces. MS Access allows users to define the relationship among tables, using visual programming.

5.1 Future Work

All participants in the study are novices on Quartz Composer and QEUP, though they received the necessary training. The study has indicated using QEUP brings positive impacts to Quartz Composer for learners. In addition, QEUP may be a potential alternative to Quartz Composer. However, for those professionals using Quartz Composer, does using QEUP still have significant impacts on their performance on Quartz Composer? Furthermore, would it be more efficient for those professionals to use QEUP instead of Quartz Composer? Future research on professional users of Quartz Composer is being considered.

A better training process could be developed. Fully understanding and motivating the use of an EUP system led to a comprehensive realization of the potential of the EUP system [7, 56, 47, 14, 65]. The learning process is affected by individual differences and the support offered [25]. As the training process improves, is the ratio of the enhancement of user performance constant for all three groups or which group would have a sharper improvement on the performance? In order to improve the training process, we could employ e-learning and collaborative learning [25]. E-learning is online learning. Instructions are provided online and users are able to achieve learning on the network. Collaborative learning is a method wherein multiple users cooperatively learn a software system. Video learning would be another preferred option.
Bibliography


Appendix A

User Study
DATE: December 1, 2009

TO: Lizao (Andy) Fang
    Computer Science

FROM: Dr. Bruce Plouffe
    Chair, Research Ethics Board

Re: End-User Programming of Quartz Composer (File # 2950910)

Please be advised that the University of Regina Research Ethics Board has reviewed your proposal and found it to be:

☐ 1. APPROVED AS SUBMITTED. Only applicants with this designation have ethical approval to proceed with their research as described in their applications. For research lasting more than one year (Section 1F), ETHICAL APPROVAL MUST BE RENEWED BY SUBMITTING A BRIEF STATUS REPORT EVERY TWELVE MONTHS. Approval will be revoked unless a satisfactory status report is received. Any substantive changes in methodology or instrumentation must also be approved prior to their implementation.

☐ 2. ACCEPTABLE SUBJECT TO MINOR CHANGES AND PRECAUTIONS (SEE ATTACHED). Changes must be submitted to the REB and approved prior to beginning research. Please submit a supplementary memo addressing the concerns to the Chair of the REB.** Do not submit a new application. Once changes are deemed acceptable, ethical approval will be granted.

☐ 3. ACCEPTABLE SUBJECT TO CHANGES AND PRECAUTIONS (SEE ATTACHED). Changes must be submitted to the REB and approved prior to beginning research. Please submit a supplementary memo addressing the concerns to the Chair of the REB.** Do not submit a new application. Once changes are deemed acceptable, ethical approval will be granted.

☐ 4. UNACCEPTABLE AS SUBMITTED. The proposal requires substantial additions or redesign. Please contact the Chair of the REB for advice on how the project proposal might be revised.

Dr. Bruce Plouffe
cc: Dr. Daryl Hepting – Computer Science

** supplementary memo should be forwarded to the Chair of the Research Ethics Board at the Office of Research Services (Research and Innovation Centre, Room 109) or by e-mail to research.ethics@uregina.ca

Phone: (306) 585-4775
Fax: (306) 585-4893
www.uregina.ca/research

Figure A.1: Ethics approval
I understand that this form and the information it contains are given to me for my protection and full understanding of the procedures. My signature on this form signifies that I understand the procedures to be used in this study and the personal risks and benefits to me in taking part, that I voluntarily agree to participate in the project, that I understand that I may withdraw my participation in this study at any time, and that my decision to participate in this study, and my subsequent involvement in it, will have absolutely no bearing on any other dealings I have with Dr. Hepting, nor the University of Regina in general.

Knowledge of my identity is not required. I will not be required to write my name or any other identifying information on the research questionnaires. Verbal comments will be encouraged during the session and will be recorded. These recordings will be used only by the researchers in reviewing your session and will not be quoted. The audio recordings will be destroyed once the researchers' review is complete. All other research materials will be held confidential by the researchers and kept in a secure location.

I agree to participate by completing a pre-task questionnaire, a task that involves the use of computer software (after appropriate training with the software), and a post-task questionnaire. I understand that these activities are not expected to require more than 45 minutes, at time to be scheduled with me and at a location on campus.

I understand that this project was approved by the Research Ethics Board, University of Regina. If I have any questions or concerns about my rights or treatment as a subject, I may contact the Chair of the Research Ethics Board at 585-4775 or by e-mail. research.ethics@uregina.ca.

I may obtain copies of the results of this study, upon its completion, by contacting Dr. Hepting, in care of the Computer Science Department, University of Regina.

I have received a copy of this consent form

NAME (please print legibly): ________________________________

SIGNATURE: ________________________________

WITNESS: ________________________________

DATE: ________________________________
PRE-TASK QUESTIONNAIRE

The following questions relate to your background and experience with computers, math and art. Please answer each question by circling the most appropriate response immediately following each question. Your answers to these questions will allow for more accurate analysis of the data collected during the study.

1. What is your gender?
   - Male
   - Female

2. How old are you?

3. How many years education have you completed past high school?
   - 0
   - 1-4
   - 5-6
   - More than 6

4. Which of the following general categories describes your main area of study/interest?
   - Fine Arts
   - Business
   - Arts/Humanities
   - Science

5. What is your level of familiarity with computers?
   - None
   - Low
   - Medium
   - High

6. How often do you create your own images/animations, including 2D and 3D?
   - Daily
   - Weekly
   - Monthly
   - Rarely
   - Never

7. What is your level of experience with computer programming?
   - None
   - Low
   - Medium
   - High

8. What is your level of experience with visual programming?
   - None
   - Low
   - Medium
   - High
   - Have not heard of visual programming

9. What is your level of experience with Quartz Composer?
   - None
   - Low
   - Medium
   - High
   - Have not heard of Quartz Composer

10. Would you describe yourself as mathematically inclined?
    - Strongly Disagree
    - Disagree
    - Agree
    - Strongly Agree

11. Would you describe yourself as artistically inclined?
    - Strongly Disagree
    - Disagree
    - Agree
    - Strongly Agree

Figure A.3: Pre-task questionnaire
Throughout your participation today, please ask if you have any questions about anything. Also, you are encouraged to talk about what you are doing as you are doing it.
Quartz Composer

Task:
In this task, you will work on Quartz Composer, which is a visual programming environment. You are given a sample file (qtz file) and you are asked to work with that sample file until you have achieved a satisfying result.

Overview of Quartz Composer:
There are FOUR windows that you may see during this study:
- **Editor** (Figure 1 left) Each rectangle is a patch, which is a black box (subsystem). You can look through patches' brief description from **Patch Creator** (Figure 3). You are freely change links among patches by dragging and dropping yellow curved lines.
- **Viewer** (Figure 1 right) Displays the real-time result.
- **Patch Inspector** (Figure 2) From drop down list on the top, select **Input Parameters** or **Settings** to access a list of parameters you may change.
- **Patch Creator** (Figure 3) This window is used to add patches. You do not have to add any new patches in this study. However, this dialogue provides a description for each patch.

Figure A.5: Tutorial page 2
cogito

Task:

In this task, you will work on cogito, which provides an interface to a graphics application. You are given a sample file (cog file). After you load the sample, your task is to work with that file until you have achieved a satisfying result.

Overview of cogito:

There are TWO main windows you will use in the study:

- **Viewer** (Figure 1) It is composed of 2*4 cells. Each cell is used to display a result. Select (by clicking the “Select” button) the results you like. You can use one or more selections define a further exploration amongst these alternatives. Click “New” (at the bottom of the window, shown in Figure 1) to do this.

- **New Space** (Figure 2) Add or remove values to be used, for each parameter, in the new exploration. The values from the selections you made will be highlighted. You can add or remove values from the lists for each parameter. In this sample file, there are four patches, each patch has a group of parameters. After finish setting parameters, click “OK.”

Figure A.7: Tutorial page 4
Start playing all animations on current screen

Navigate to Previous or Next Screen

Start new exploration, pop up New Space dialog

Sort results by setting a key

Figure 1 Viewer

Figure A 8 Tutorial page 5
A list of values. Values with blue background represent the values being selected. Values with prefix "patch_" means the values are the outputs of other patch.

Figure 2 New Space
End-user programming

Tasks:
In this study, you will work with an end-user programming editor (eupe) that provides an interface to a graphics application. You are given a sample file (qtz file). In this task, you will work on cogito, which provides an interface to a graphics application. You are given a sample file (qtz file). After you load the sample, your task is to work with that file until you have achieved a satisfying result.

Overview of End-user programming of Quartz Composer:
There are FOUR main windows you will use in the study:

- **Editor** (Figure 1) The main window to display patches, parameters, and values. When you double-click a patch, the system will load its description into the description panel. The interface to edit a description is shown in Figure 3. The editor will allow you to define which parameter values you want to explore.

- **Viewer** (Figure 2) It is composed of 2×4 cells. Each cell is used to display a result. Select the results you like. Transfer selected results to Editor by clicking “Explore”.

- **Edit Description** (Figure 3) Define and customize description of patches and their parameters.

- **Add Values** (Figure 4) Add or remove values for each parameter. You can add values from three categories: pre-defined list of values, manually inputting a new value, and other external patches according to their types, which can be defined in Edit Description (Figure 3).

All patches in the sample need descriptions, and you will have to complete some of them. You may choose to customize the description of a patch and its parameters using the “Edit Description control (Figure 3). The description includes some important information that would probably affect the result generated. It is recommended to look through the whole description before your adding values.

Figure A.10: Tutorial page 7
**Build** process input values and generate results in *Viewer*

**Viewer** Navigate to *Viewer*

**Patches ListBox** list all patches in the file. You need define patches' description first if the patches have no tick icon.

Display the description of selected node and its parameters.

Click 'Edit' (at bottom right of description) to edit description.

Add values to parameters’ The system will pop up *Add Values* Dialog.

Delete remove selected value from the list.

History: Generate a history record by clicking Build.

Select a record click Load ' to load a history record.

Figure 1 Editor
**Editor** Navigate to Editor

**Play** Start playing all animations on current screen

**Explore** Transfer selected results to Editor, to continue exploration

---

**Figure 2 Viewer**

---

Figure A 12 Tutorial page 9
Blibord also has several unique features.

Both dimensions \((x, y)\) of the quad can be set directly.

Figure 3 Edit Description

Figure 4 Add Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3</td>
<td>left</td>
</tr>
<tr>
<td>0.7</td>
<td>leftline</td>
</tr>
<tr>
<td>-1</td>
<td>leftline</td>
</tr>
<tr>
<td>0</td>
<td>center</td>
</tr>
<tr>
<td>0.3</td>
<td>right</td>
</tr>
<tr>
<td>0.7</td>
<td>rightline</td>
</tr>
</tbody>
</table>
POST-TASK QUESTIONNAIRE

Please answer each question by circling the most appropriate response immediately following each question. Your answers to these questions will allow for more accurate analysis of the data collected during the study.

Quartz Composer

1. I felt comfortable while I was using Quartz Composer to finish the tasks
   - Strongly Disagree
   - Disagree
   - Agree
   - Strongly Agree

2. Tasks become quite easy if I can do work based on a sample
   - Strongly Disagree
   - Disagree
   - Agree
   - Strongly Agree

3. If there was not a sample offered, it would be hard for me to figure out how to use Quartz Composer
   - Strongly Disagree
   - Disagree
   - Agree
   - Strongly Agree

4. I clearly understood the workflow of the sample provided
   - Strongly Disagree
   - Disagree
   - Agree
   - Strongly Agree

5. Quartz Composer provides enough help information (including all text information, such as help document, tooltip shown up when your mouse hover on objects, etc.)
   - Strongly Disagree
   - Disagree
   - Agree
   - Strongly Agree

6. I clearly understood the help information offered by Quartz Composer
   - Strongly Disagree
   - Disagree
   - Agree
   - Strongly Agree

7. Most time, when I changed values of parameters, the results shown on Viewer were what I expected
   - Strongly Disagree
   - Disagree
   - Agree
   - Strongly Agree

8. What did you like most about Quartz Composer? Why?

9. What did you like least about Quartz Composer? Why?

10. In your opinion what is necessary help information that Quartz Composer is supposed to provide?

Figure A.14: Post-task questionnaire page 1
Cogito

1 I felt comfortable while I was using Cogito to finish the tasks
   Strongly Disagree Disagree Agree Strongly Agree
2 I clearly understood the workflow of the sample provided
   Strongly Disagree Disagree Agree Strongly Agree
3 Displaying 8 results each time is more helpful than that displaying 1 result each time
   Strongly Disagree Disagree Agree Strongly Agree
4 New Space Dialog is helpful and easy to understand
   Strongly Disagree Disagree Agree Strongly Agree
5 The functionality to sort results by a certain parameter is helpful (List box on the right bottom)
   Strongly Disagree Disagree Agree Strongly Agree
6 What did you like most about Cogito? Why?
7 What did you like least about Cogito? Why?
8 Can you suggest any changes to the design of Cogito?
End User Programming (EUP)

1. I felt comfortable while I was using EUP to finish the tasks
   - Strongly Disagree
   - Disagree
   - Agree
   - Strongly Agree

2. I clearly understood the workflow of the sample provided by the author
   - Strongly Disagree
   - Disagree
   - Agree
   - Strongly Agree

3. It is helpful to provide description for each node (patch) and each parameter
   - Strongly Disagree
   - Disagree
   - Agree
   - Strongly Agree

4. It is helpful to allow users to define the type of each node (patch)
   - Strongly Disagree
   - Disagree
   - Agree
   - Strongly Agree

5. It is helpful to allow users to define the validation of each parameter
   - Strongly Disagree
   - Disagree
   - Agree
   - Strongly Agree

6. It is helpful to allow users to customize the name of each node (patch) and parameter
   - Strongly Disagree
   - Disagree
   - Agree
   - Strongly Agree

7. It is helpful to display parameters values as readable text
   - Strongly Disagree
   - Disagree
   - Agree
   - Strongly Agree

8. It is helpful to allow users to select values from 3 different categories
   - List of Values
   - Customized Values and Patches
   - Strongly Disagree
   - Disagree
   - Agree
   - Strongly Agree

9. What did you like most about EUP? Why?

10. What did you like least about EUP? Why?

11. Can you suggest any changes to the design of EUP?
B.1 Quartz Composer, open-ended comments

Listed below are open-ended user comments provided by users in their response to Question 8 in Quartz Composer post-task in Appendix A.

Question 8. What did you like most about Quartz Composer? Why?

- The result is showing instantly in the Viewer which helps me revise the parameters.

- I like that it provided realtime updating, seeing directly how the patches interfaced was helpful.

- I liked the drag and drop connections between elements.

- The ability to drag a connection line from one patch to another patch property. Seeing a graphical connection helped me understand what was happening.

- Dragging wires to new nodes. Very easy and intuitive.

- I liked the way that you could connect variables to the image. Made it easy to know what was pointing to what.

- The real-time result which can help determine the result of the change I made.
- I can directly see the changes I have made. And it’s easy to add new features to the animation.

- I like the 2D patch, it makes easier for user to operate.

- The drag and link function was nice and useful.

- The real-time display and visual patching.

- It’s easy to use. Very flexible program.

- There were a lot of things I was able to change. I was able to combine different patches to make some really cool effects.

- You can see the results in real-time.

- I liked the visual representation of how a value is put in effect because it illustrated best how one thing affects another.

Listed below are open-ended user comments provided by users in their response to the Question 9 in Quartz Composer post-task in Appendix A.

Question 9. What did you like least about Quartz Composer? Why?

- If you created too many patches, the screen get a bit cluttered.

- Clicking on the dot to change values was not intuitive. Also did not like that there was no easy way to reset changes to values and that try stayed changed if you tried something.

- That I could not copy a patch. It would be nicer than going to patch creator every time. (It may be there is a menu, I only looked in the right-click menu)

- List of input parameters was confusing.

- I had found there were some issues with linking objects. It also disabled my image without telling me. As well as instructions on how to do layers. The reasoning it made me feel like I made a mistake.
• I was not used to some of the functionality thus when trying to have a dynamic background, I wouldn’t figure out how.

• Still not enough freedom for users to play with. Comparing to Flash, the objects has relatively few attributes to modify.

• The parameters some time confused.

• Much harder to figure out which variables to change. (i.e. patch info -i, settings or variables? )

• Rotation is in degrees. So for some users it may be confusing.

• At first it was very confusing. Also, I have a feeling that if I had a specific effect in mind that I wanted to create. It would be difficult to know what to change.

• The colors could not be randomly chosen.

• I disliked how foreign it was to me.

Listed below are open-ended user comments provided by users in their response to Question 10 in Quartz Composer post-task in Appendix A.

Question 10. In your opinion what is the necessary help information that Quartz Composer is supposed to provide?

• Basic instruction on what each patch does.

• what each values does and how changing that value affect the image.

• If it did not have it, a tooltip to explain what each parameter is for.

• Initial introduction describing the interface.

• A better description in the hover option. There was a lot of times I would make a change and not know what happened.
• They already provided quite a bit of information, such as if a component is suitable for each connection as well as description for each when creating as new patch, but maybe simpler categorization may help novice users.

• More public variables for users to change.

• Some function's parameters, like sin, cos, etc. Should given the path way and range.

• Maybe a tutorial/walkthrough of features.

• Should be more flexible in axis. Adjusting them by user, comparing the old and the new image, etc.

• Perhaps it could organize the patches in a way I could understand more easily. I didn't really know what a “filter” was until I hooked one up.

• It is very well supplied, and help was not needed.

• I think if it gave a tutorial on the computer that worked in time with the user it would be beneficial.

B.2 cogito, open-ended comments

Listed below are open-ended user comments provided by users in their response to Question 6 in cogito post-task in Appendix A.

Question 6. What did you like most about Cogito? Why?

• It allowed you to create a large amount of animations at once and compare them against each other. That made it easier to experiment.

• The play button because I understand what it did.

• I loved how it would display the different types of images and the variables to select what you wanted. Very easy to navigate. To manipulate the movement of the graphic was easy. Also the provided instruction for what each movement meant.
• You can see many different results at the same time.

• Seeing many different results allowed me to compare the changes I had made.

Listed below are open-ended user comments provided by users in their response to Question 7 in cogito post-task in Appendix A.

Question 7. What did you like least about Cogito? Why?

• An output for the random value would be very helpful. To narrow down what you want displayed.

• The list view for patch properties. It was difficult to find specific properties in a timely fashion.

• The color selection. While it has the basics would be nice to have a pinwheel of the color filter. Because it would be nice to be able to get more color variation other than the regular spectrum of RGB.

• Some parameters are little hard to understand.

• There weren’t as many characteristics of Billboard that I could modify.

Listed below are open-ended user comments provided by users in their response to Question 8 in cogito post-task in Appendix A.

Question 8. Can you suggest any changes to the design of Cogito?

• Maybe change the new space dialogue box to a vertical one with drop-down boxes.

• Place a filter in the properties window so not all properties are shown at once.

• The pin wheel for color selections and as well as instead of having all options lay out horizontal, have a few, like five. Then have the next five in the row down and so forth.

• Maybe the parameters can have its own animation examples for users to understand.

• Add more functionality to it. More modifiable options.
B.3 QEUP, open-ended comments

Listed below are open-ended user comments provided by users in their response to Question 9 in QEUP post-task in Appendix A.

Question 9. What did you like most about EUP? Why?

- Multiple parameters selection is provided. Customer can select the ideal result according to the comparison.
- Being able to see the permutation of parameter selections.
- It felt very customizable, like a programming language without syntax.
- The ordering system made sense to me more i.e. parameters inside patches are like files inside of files.

Listed below are open-ended user comments provided by users in their response to Question 10 in QEUP post-task in Appendix A.

Question 10. What did you like least about EUP? Why?

- The result can be explored to Editor for the revision of preference.
- Navigating between patches and their parameters easy to get lost.
- After select results have to go back each time. I don’t like to go backward forward.
- It was a little confusing to get into at first. There wasn’t real-time image display.
- It was more difficult for me to figure out what action led to what reaction.

Listed below are open-ended user comments provided by users in their response to Question 11 in QEUP post-task in Appendix A.

Question 11. Can you suggest any changes to the design of EUP?

- It might be helpful if more validation parameters ranges can be provided.
- Interactive sliders for changing values with live feedback. Perhaps a try view for patches, parameters, values.

- Adding real-time displays and a more visual patch scheme would be nice.

- If the image on the main screen changed as you worked through it.