Assessing End-user Programming for a Graphics Development Environment

Lizao Fang and Daryl H. Hepting

Computer Science Department, University of Regina, Canada
fang205l@uregina.ca, dhh@cs.uregina.ca

Abstract. Quartz Composer is a graphics development environment that 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, especially by end-users. This paper presents a prototype end-user programming system that is designed to remove the barriers present in the native Quartz Composer environment. 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. In order to assess the impacts of QEUP, a user study was conducted with 15 participants. Preliminary results indicate that there may be benefit to using QEUP when first learning Quartz Composer, or when learning new capabilities within it.

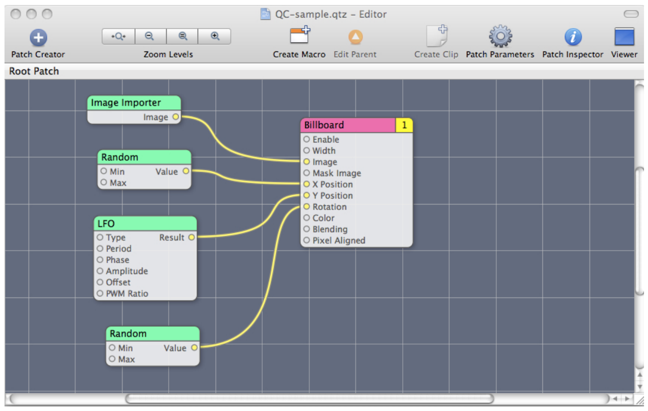
1 Introduction

The power of the visual programming paradigm, which began with ConMan [1], is evident in the Quartz Composer graphics development environment (GDE) now available on Apple computer systems. This direct-manipulation style of programming makes clear the relationships between different parts, called patches, and it affords alteration of those relationships by changing the connections between patches. See Figure 1 for a view of the programming interface. Other modifications are also possible through other features of the interface.

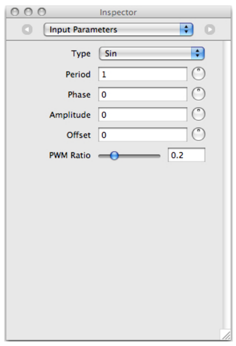
Yet, if the user would like to explore different variations possible from this simple program comprising the 5 patches in Figure 1, he or she must be responsible for all of the changes. This necessity leads to two questions that are of interest here:

1. given an unsatisfactory output, which changes can be made to improve it?
2. after evaluating several alternatives, how can one easily return to, or reprogram, the best output amongst them?

Following from the classification developed by Kochhar et al. [2], graphics development environments may be classified as either manual, automated, or augmented. Manual systems require complete involvement of a human to construct a graphics application; automated systems require no involvement of a



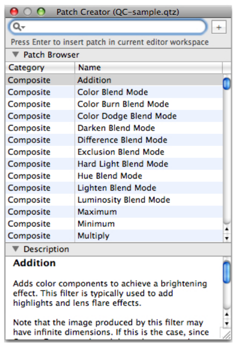
(a) Editor



Input Parameters

Type	Sin
Period	1
Phase	0
Amplitude	0
Offset	0
PWM Ratio	0.2

(b) Patch Inspector


Patch Creator (QC-sample.qtz)
Press Enter to insert patch in current editor workspace
Patch Browser
Description

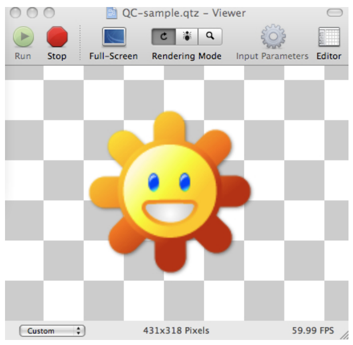
Category	Name
Composite	Addition
Composite	Color Blend Mode
Composite	Color Burn Blend Mode
Composite	Color Dodge Blend Mode
Composite	Darken Blend Mode
Composite	Difference Blend Mode
Composite	Exclusion Blend Mode
Composite	Hard Light Blend Mode
Composite	Hue Blend Mode
Composite	Lighten Blend Mode
Composite	Luminosity Blend Mode
Composite	Maximum
Composite	Minimum
Composite	Multiply

Addition

Adds color components to achieve a brightening effect. This filter is typically used to add highlights and lens flare effects.

Note that the image produced by this filter may have infinite dimensions. If this is the case, since

(c) Patch Creator



(d) Viewer

Fig. 1. Quartz Composer, with the sample used throughout this paper.

human; and augmented systems support some notion of the development process as a collaborative effort between human and computer.

Quartz Composer, with its visual programming paradigm, is a manual environment. This paper describes the development and testing of an augmented system, called QEUP (for Quartz Composer End-user Programming) which is based on *cogito* [3]. As well, QEUP is an end-user programming system, which can be used by end-users with little or no programming experience. According to the spectrum of software-related activities proposed by Fischer and Ye [4], the entry level of visual programming is higher than that of end-user programming. Can additional supports for end-user programming remove the barriers that remain with Quartz Composer, highlighted by the two questions posed earlier?

The rest of this paper is organized as follows. Section 2 presents some background work. Section 3 describes the software that is being studied. Section 4 describes the design of the user study. Section 5 describes the results of the user study. Section 6 presents some conclusions and opportunities for future work.

2 Background

Although visual programming systems do enable end-user programming to some extent, the differences between these two areas are highlighted here.

2.1 Visual Programming

Visual programming employs pictorial components to specify programs. In many visual programming environments (VPEs), these pictorial components are connected according to the required data flow. In some other systems, the pictorial components are constructed based on flowcharts. Many VPEs allow users to drag and drop these components. The most exciting feature of visual programming is not the colourful graphical components, but its ability to develop, test and modify in an interactive way [5]. The barriers to visual programming are lower than conventional programming [6], and everyone with some necessary training would be able to 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 approximately real-time, computing. Furthermore, some VPEs such as FPL (First Programming Language) [7] and Quartz Composer are well-suited for end-users because the systems eliminate syntactic errors [6]. Some other samples of VPEs include: ConMan [1], LabVIEW [8], Alice [9], and Scratch [10].

However, visual programming has not been widespread. Some researchers [6, 11, 12] point out reasons, which are summarized as follows:

– visual programming is not radical enough to describe dynamic processes
– pictorial components in visual programming increase abstraction, because they are symbolic objects
– pictorial components waste precious screen real estate
– visual programming inhibits details
– visual programming does not scale well
– visual programming has no place for comments
– visual programming is hard to integrate with other programming languages, such as text.

2.2 End-user programming

End-user programming (EUP) is defined as the activities that end-users, with no or minimal programming knowledge, create functions or programs. EUP also is called end-user modifiability [13, 14], end-user computing [15], and end-user development [16].

The proposed scope of EUP varies amongst different researchers. Ye and Fischer [4] defined EUP as activities performed by pure end-users, who only use software rather than develop software. However, Blackwell [17] categorized EUP into five categories: activities based on a scripting language; activities performed in visual programming environments; activities performed in graphical rewrite systems; activities relying on spreadsheet systems; and activities with example-based programming. The scope of EUP brought forward by Myers, Ko and Burnett [18] covers almost all software-related activities.

End-users of EUP systems are those people having limited programming knowledge. The entry level into EUP systems is supposedly relatively low. We propose the following requirements for EUP systems - in the context of GDEs - to answer the question of which software systems support EUP.

1. *support for creative activities*: designers cannot envision the results created by end-users. The system enables end-users to create their own programs.
2. *ordinary end-users are able to benefit*: the only requirement could be that users have experience of using computers. Some domain-oriented systems, such as MatLab [19], require solid domain knowledge that makes them inaccessible outside of the target domain. In contrast, Microsoft Excel is widely accepted by users from various domains.
3. *easy to learn and easy to use*: many end-users may not have the patience to learn a complex system. They do not have much time, nor do they want to spend a lot time learning software applications. After a short training period, end-users are able to recognize and comprehend a large percentage of functions provided by the EUP systems.
4. *fault-tolerant and interactive*: the system should render results without crashing, even though the results could be unreasonable. Maintaining responsiveness will help to engage the end-user.

3 Quartz Composer, *cogito* and QEUP

Three different software packages were involved in this paper. Each of them is described in more detail here: Quartz Composer, *cogito*, and QEUP. For each section, a usage scenario describes how a sample, which presents the movement of a cartoon sun, can be refined in each system. The actor in each scenario, named Jimmy, is an end-user with little programming experience. In addition, the implementation of QEUP is described.

3.1 Quartz Composer

Quartz Composer is a visual programming environment as well as a graphics development environment. There are four main windows in Quartz Composer (See Figure 1): the Editor (Figure 1(a)) window for constructing the program; the Patch Inspector (Figure 1(b)) window for setting specific parameter values; the Patch Creator (Figure 1(c)) window for adding new patches to the program; the Viewer (Figure 1(d)) window for displaying the output in real-time.

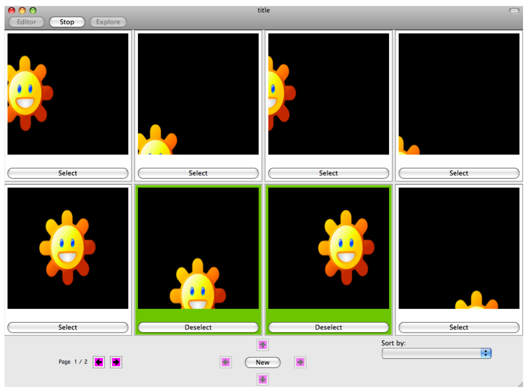
To perform programming, end-users drag and drop patches then connect them by dragging lines from source ports to destination ports. The values produced from the source patch are passed to the destination patch through their ports.

Usage Scenario

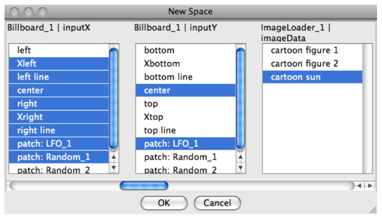
Step 1: Jimmy opens the example file, the Editor and Viewer windows pop up. In the Editor window (Figure 1(a)), he sees 5 boxes connected by lines.
Step 2: The cartoon sun is moving in the horizontal direction, and Jimmy wants to add a movement effect in the vertical direction. He selects the LFO patch and clicks the "Patch Inspector" to access the Patch Inspector window. Jimmy changes the value of the Amplitude from 0 to 0.2.
Step 3: Jimmy removes the horizontal movement effect by disconnecting the line between the Random patch (top) and the Billboard patch.
Step 4: To add new patches to the program, Jimmy drags new patches from the Patch Creator window to the Editor window.
Step 5: Jimmy refines the parameter values for the new patches, as described above, to produce a satisfying animation.

3.2 *cogito*

Hepting [3] developed *cogito*, which was designed to address drawbacks of traditional visualization tools. In this case, *cogito* presents 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(b)) dialogue box, and generate new alternatives for consideration on the Viewer (Figure 2(a)) window.



(a) Viewer



Billboard_1 | inputX	Billboard_1 | inputY	ImageLoader_1 | imageData
left	bottom	cartoon figure 1
✓ Xleft	Xbottom	cartoon figure 2
✓ left line	bottom line	✓ cartoon sun
✓ center	✓ center	
✓ right	top	
✓ Xright	Xtop	
✓ right line	top line	
✓ patch: LFO_1	✓ patch: LFO_1	
✓ patch: Random_1	patch: Random_1	
patch: Random_2	patch: Random_2	

(b) New Space

Fig. 2. *cogito*'s user interfaces

Usage Scenario

Step 1: Jimmy opens the example file, which is then shown in the first cell of the Viewer window (Figure 2(a)).

Step 2: Jimmy selects the animation in the first cell and the background of the cell turns green. He clicks the "New" button on the bottom, then accesses the New Space dialogue box, which shows what he has selected so far. Jimmy clicks to add values in the "Billboard_1 | inputX" and "Billboard_1 | inputY".

Step 3: He clicks the "OK" button and eight new animations are displayed in the Viewer window (Figure 2(a)). Two of them are interesting, so Jimmy selects them and the backgrounds of their cells turn green as well.

Step 4: Jimmy navigates between screens by clicking the arrow buttons and continues selecting other appealing outputs.

Step 5: Jimmy clicks the "New" button again to refine his selections. He continues to explore until he decides on his favourite animations.

3.3 QEUP

QEUP is an example-based end-user programming system. Users start with an example, following the bottom-up recursive learning method. Specific examples aid visualization [20] and help users create concrete mental models. Unlike Quartz Composer, which is a manual system, QEUP is an augmented system. In Quartz Composer, users have to track the value of each parameter in order to identify its effect on the final result. 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 and selecting outputs from diverse alternatives.

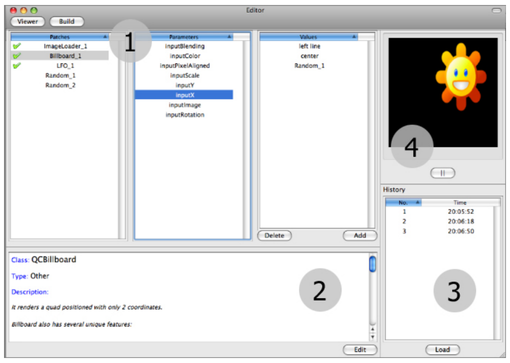
Usage Scenario

Step 1: Jimmy selects the example file. The Editor (Figure 3(a)) window is displayed and the example animation is shown (Figure 3(a), circle 4). Patch names are listed in the Patches control list (Figure 3(a), circle 1), along with their parameters and values. Jimmy realizes that new patches must be defined to the system from the Description Configuration dialogue box (Figure 3(b)) before he can explore their parameter values.

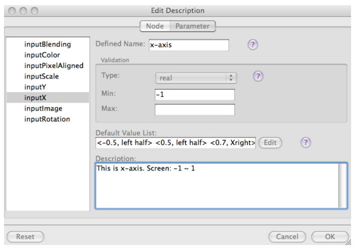
Step 2: Jimmy begins to edit patch descriptions. These descriptions include enforceable constraints on value ranges and types. Jimmy realizes that all these activities are helping him to learn about the example, and he becomes more confident that he could tackle more complicated configurations quite easily. After finishing the description of a patch, Jimmy realizes that his description is displayed in the Editor window (Figure 3(a), circle 2).

Step 3: In the Editor window, he navigates to a parameter, and clicks the "Add" button under the Values control list. In the Add Values (Figure 3(d)) dialogue box that pops up, he realizes that the values he added to his description appear as defaults under the Value List. He sees that he can also input new values manually, or use other patches to generate values. He doesn't change anything at this time.

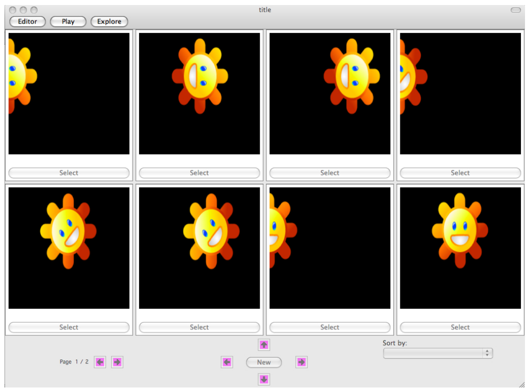
Step 4: Jimmy clicks "Build" button in order for the system to produce the outputs, based on his choices. In the Viewer (Figure 3(c)) window, he reviews the outputs and selects those which he finds interesting.



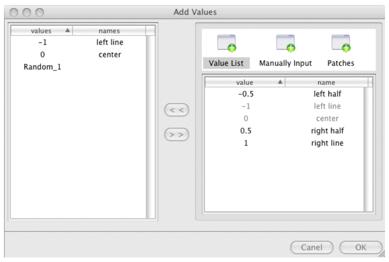
(a) Editor



(b) Description Configuration



(c) Viewer



(d) Add Values

Fig. 3. QEUP's user interfaces

Step 5: By clicking the "Explore" button, Jimmy transfers his selections' data back to the Editor window, where he continues to refine his choices. He is confident about making changes here, because he knows that he is able to load a history of his previous work (Figure 3(a), circle 3).

The implementation of the QEUP functionality is divided into four phases. XML is a cornerstone of the system because most documents involved are in XML format.

Phase 1: After the example file (.qtz) is successfully loaded, the system converts the document from binary property list to XML format. The XML document is processed and related information, such as patch names, parameter names, and values, is picked up from the document. As well, the Description document (*eup_description.xml*) is processed. New patches that do not have descriptions are identified by the system.

Phase 2: When users write the description for the new patches, the description information is saved into *eup_description.xml*, which will be processed again if the users edit the description.

Phase 3: The system begins to produce outputs when the Build button on the Editor window is clicked. During the processes of generating outputs, at first the Template document (*eup_standarizedSample.xml*) is produced based on the XML format of the example document. The Template document inherits the XML structure of the example document, but removes those data which vary in different outputs. This Template document is created only once for each example file. A *cog* document is used to save the parameters and their values. The system processes and generates an XML scheme document that includes any restrictions defined by users. Then, the XML scheme document is employed to validate data in *cog* document. If the validation succeeds, the data in the *cog* document constructs a set of combinations, each of which will be used to produce an output. Based on the Template document, the first eight combinations are processed. The system produces eight outputs and puts them in the Viewer window.

Phase 4: Other outputs will be dynamically produced when users navigate to the next screen. Users might select several satisfying outputs and transfer them to the Editor window for further exploration. During this process, the system maps the selected outputs to corresponding combinations mentioned in Phase 3. Then, the values of the parameters are appended to the Values list in the Editor window.

4 User Study

A user study was conducted in order to assess the impacts of QEUP for Quartz Composer. The between-subjects study was designed to look at the users' experience with Quartz Composer in three different cases.

4.1 Participants

15 participants took part in the study, with ages ranging from 18 to 32. All of them were students at the University of Regina. Their areas of study were diverse, but most had taken at least one Computer Science course. Regarding their level of programming knowledge, 4 reported a low level; 5 reported a medium level, and 6 reported a high level. 14 participants reported no experience with Quartz Composer, and 1 participant reported low experience on Quartz Composer. They have at most medium experience on visual programming.

The participants were randomly grouped assigned to 3 groups. Participants in the first 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).

4.2 Materials and Task Design

Each participant also encountered the following documents: a pre-task questionnaire, which covered aspects of participants' background; a tutorial manual, which provided a standard introduction to the software systems being used; and a post-task questionnaire, which captured aspects of their experience with the software system(s) used by the participant.

During each part of the study, participants began with a very simple example (shown in Figure 1). It only had 5 patches: an Image Importer (to load the image), a Billboard (to display the image), an LFO (low-frequency oscillator) and 2 Random patches, used to move the Billboard around the screen. Participants were requested to work on the input example, refine the example, and produce appealing outputs.

4.3 Procedure

Participants completed a consent form and the pre-task questionnaire. The QC Group received some training and then used Quartz Composer, for a period of at most 15 minutes. The *cogito* Group received some training and then used *cogito* followed by Quartz Composer, each for a period of at most 15 minutes. The QEUP Group received some training and then used QEUP, followed by Quartz Composer, each for a period of at most 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 the 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).

5 Results and Analysis

We analyzed all participants' performance on Quartz Composer. All data in following tables in this section is collected from performance on Quartz Composer. The participants' performance is analyzed from three aspects: time to complete the task, attempts to set values, and the primary operation performed by participants. The final outputs in Quartz Composer are determined by parameters' values and the connection relationship amongst patches.

Time to complete the task In the study, the time spent on Quartz Composer was limited to 15 minutes, but participants were able to stop before that time. Table 1 shows the time spent by all participants from all three groups on Quartz Composer. The total and average time spent is shortest for the QEUP Group.

All participants are beginners with respect to Quartz Composer. But participants using QEUP seemed able to more efficiently produce satisfying outputs within Quartz Composer compared to other participants. Users communicate with computers based on two channels: explicit and implicit [21]. The explicit channel is based on the user interface and the implicit channel relies on the knowledge that the users and the computers both have. Because the participants in this study have no, or very limited, experience on Quartz Composer, their communication with the computer through the explicit channel is not very different. Therefore, the improvement of the communication through the implicit channel may be the factor that results in less time spent in the QEUP Group. The QEUP Group might have gained necessary knowledge after they have some experience on the QEUP system.

Table 1. Time to complete task (min)

Group	Participant Times					Total	Avg.
QC	10.00	13.28	15.00	15.00	15.00	68.28	13.66
cogito	14.25	15.00	15.00	15.00	15.00	74.25	14.85
QEUP	9.22	10.25	11.00	11.75	15.00	57.22	11.44

Attempts to set values All participants performed programming based on the example. In order to set a suitable value for a parameter, participants might try several times. We calculated the ratio of set-value attempts made to the set-values attempts kept, which is meant to represent how many trial values were needed in order to successfully customize a parameter. We wanted to evaluate impacts of exposure to *cogito* and QEUP on Quartz Composer performance. The data in this table is the data related to the example only, operations on new patches were not considered. Table 2 shows data from the three groups. The minimum ratio is 1 and so it is reasonable that this is the ratio for participants who did not set any values.

The QEUP Group has the smallest ratio, which may indicate that use of the QEUP system helps participants to make better decisions in Quartz Composer. The significance of using *cogito* on this aspect is less obvious. As well, Table 2 supports the assertion that the participants' knowledge of Quartz Composer in the QEUP Group is improved by using QEUP first.

Table 2. Setting values on sample

Group		User Performance					Avg.
QC	Set-value operations	28	30	40	14	30	
	Set-value kept	20	20	22	7	10	
	Ratio	1.40	1.50	1.82	2.00	3.00	1.94
cogito	Set-value operations	0	17	26	29	61	
	Set-value kept	0	13	17	13	27	
	Ratio	1.00	1.31	1.53	2.23	2.26	1.67
QEUP	Set-value operations	0	21	9	18	16	
	Set-value kept	0	17	7	12	10	
	Ratio	1.00	1.24	1.29	1.50	1.60	1.32

Primary operation: setting values or connection/disconnection Table 3 provides the number of set-value and connection/disconnection operations attempted, from which a ratio is calculated. If ratio is great than 1, setting values seems to be the primary operation. If it is less than 1, connecting/disconnecting patches is said to be the primary operation. The QC Group has no participants whose primary operation was connecting/disconnecting patches. However, there is 1 in the *cogito* Group and 3 in the QEUP Group.

In the QC Group, we observed that participants expended much effort on tracking parameters' values. However, participants in *cogito* Group and QEUP Group tend to be less concerned about tracking parameters' values. Patch Inspector (Figure 1(b)) window for setting values is a window on the second level in Quartz Composer. It is accessible by clicking Patch Inspector button on the Editor Window. Connection/disconnection is the 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 have capabilities to 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.

Table 3. Setting values and connection/disconnection

Group		User Performance				
QC	Set-value operations	30	28	45	14	30
	(Dis)connection operations	29	26	30	5	6
	Ratio	1.03	1.08	1.50	2.80	5.00
cogito	Set-value operations	16	17	33	29	61
	(Dis)connection operations	52	17	21	9	14
	Ratio	0.31	1.00	1.57	3.22	4.36
QEUP	Set-value operations	0	9	21	18	18
	(Dis)connection operations	29	13	26	6	2
	Ratio	0.00	0.69	0.81	3.00	9.00

6 Conclusion and Future Work

From analysis of the user study, it appears that QEUP may be a useful complement to Quartz Composer, which may help end-users to acquire knowledge and to form mental models. Those participants starting with QEUP, followed by Quartz Composer, seemed better able to cope with the barriers that emerge in Quartz Composer. It may be that experience with QEUP can provide a headstart in learning the Quartz Composer graphical development environment. Furthermore, it may provide a similar benefit when an end-user wishes to better understand new features and patches within Quartz Composer. A longer-term study would be needed to better assess both potential outcomes hinted at from these results. As well, a comprehensive study would be needed to evaluate whether the QEUP system could be an alternative to replace Quartz Composer.

Acknowledgements. The authors wish to acknowledge support from the Natural Sciences and Engineering Research Council of Canada, the Canada Foundation for Innovation, the Saskatchewan Innovation and Science Fund, and the University of Regina.

References

1. Haeberli, P.E.: Conman: a visual programming language for interactive graphics. In: Proc. SIGGRAPH '88, New York, NY, USA, ACM (1988) 103–111
2. Kochhar, S., et al.: Interaction paradigms for human-computer cooperation in graphical-object modeling. In: Proc. Graphics Interface '91. (1991) 180–191
3. Hepting, D.: Towards a visual interface for information visualization. In: Proc. Information Visualisation 2002. (2002) 295–302
4. Ye, Y., Fischer, G.: Designing for participation in socio-technical software systems. In: Universal Access in HCI, Part I, LNCS 4554. (2007) 312–321
5. Burnett, M., et al.: Toward visual programming languages for steering scientific computations. IEEE Comput. Sci. Eng. **1**(4) (1994) 44–62
6. Myers, B.A.: Taxonomies of visual programming and program visualization. J. Vis. Lang. Comput. **1**(1) (1990) 97–123
7. Cunniff, N., et al.: Does programming language affect the type of conceptual bugs in beginners' programs? SIGCHI Bull. **17**(4) (1986) 175–182
8. Johnson, G.W.: LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control. McGraw-Hill School Education Group (1997)
9. Pierce, J.S., et al.: Alice: easy to use interactive 3D graphics. In: Proc. UIST '97, New York, NY, USA, ACM (1997) 77–78
10. Resnick, M., et al.: Scratch: programming for all. CACM **52**(11) (2009) 60–67
11. Kahn, K.: Drawings on napkins, video-game animation, and other ways to program computers. CACM **39**(8) (1996) 49–59
12. Brooks, Jr., F.P.: No silver bullet essence and accidents of software engineering. Computer **20**(4) (1987) 10–19
13. Fischer, G., Girgensohn, A.: End-user modifiability in design environments. In: Proc. CHI '90, New York, NY, USA, ACM (1990) 183–192
14. Girgensohn, A.: End-user modifiability in knowledge-based design environments. PhD thesis, University of Colorado at Boulder, Boulder, CO, USA (1992)
15. Brancheau, J.C., Brown, C.V.: The management of end-user computing: status and directions. ACM Comput. Surv. **25**(4) (1993) 437–482
16. Dörner, C., et al.: End-user development: new challenges for service oriented architectures. In: Proc. WEUSE '08, New York, NY, USA, ACM (2008) 71–75
17. Blackwell, A.F.: Psychological issues in end-user programming. In: End User Development, Springer Netherlands (2006) 9–30
18. Myers, B.A., Ko, A.J., Burnett, M.M.: Invited research overview: end-user programming. In: Proc. CHI '06, New York, NY, USA, ACM (2006) 75–80
19. Gilat, A.: MATLAB: An Introduction with Applications. New Age (2005)
20. Lieberman, H.: An example based environment for beginning programmers. Instructional Science **14**(3-4) (1986) 277–292
21. Fischer, G.: User modeling in human–computer interaction. User Modeling and User-Adapted Interaction **11**(1-2) (2001) 65–86