Integrate Processing into Exhibition Contents Authoring System

Songlin Piao, Jae-ho Kwak, Jong-Min Hyun, Whoi-Yul Kim
Department of Electrical and Computer Engineering
Hanyang University
Seoul, Korea 133–791
Email: {slpiao,jhkwak,jmhyun}@vision.hanyang.ac.kr
wykim@hanyang.ac.kr

***Abstract*—The procedure of integrating *processing*, a famous open source programming language and integrated development environment (IDE) built for the media artists, into Exhibition Contents Authoring System (ECAS) is presented in this article. The detailed architecture of *Processing* and ECAS is analyzed. The final system is implemented using Eclipse Modeling Framework (EMF) and Graphical Modeling Framework (GMF) based on Eclipse Rich Client Platform. With this tool, artists could present their works written in *Processing* easily inside ECAS.**

***Index Terms*—Media Art, Authoring, GMF, Eclipse, RCP, Processing, Plug-in.**

I. Introduction

Processing is an open source programming language and IDE built for the electronic arts and visual design communities with the purpose of teaching the basics of computer programming in a visual context, and to serve as the foundation for electronic sketchbooks [1], [2]. Ecas was developed in order to provide easier and faster way to compose media art for the media artists who do not know how to program even a single line of code [3]. All they need to do is just drop and drag components of their choice, and then just connect these components in particular sequence by drawing lines between them. ECAS was developed using graphical modeling framework [4]. As one of the most popular media art composition tools, *Processing* has already provided plenty of high quality examples. It is necessary to integrate *Processing* functionality into ECAS in order to reuse so many already existing resources.

A. Related Work

It is becoming easier and faster for the artists to create their works thanks to the development of computer technology. But there are still some limitations. One of them is that they still have to write programming code to some degree. However, most of professional artists do not know how to program even a single line of code. Our software Exhibition Contents Authoring System (ECAS), a media art contents authoring tool, has been developed for those artists in mind so that they could create their media art contents very easily.

There are several media art contents authoring tools available on the market in the form of commercial software or open source software. Max/MSP [5] and *Processing* [2] are two of the most widely used programs all over the world. Max was originally written by Miller Puckette as the patch editor for the Macintosh [1]. Then it was further developed by third parties to extend its functions gradually. There is an open source version of Max named Pure Data [6]. Pure Data, also developed by Miller, is an alternative tool for the students who learn digital music. *Processing* was firstly developed by the MIT Media lab in order to help out the artists who suffered from programming code. The development began in 2001 and the stable version 1.0 was firstly released by the year 2008. There are many media art contents developed and implemented using *Processing*. Fig. 1(a) and Fig. 1(b) show the snapshots of ECAS and *Processing*, respectively.

Besides these programs, there are also other media art contents authoring tools like VVVV [7], Quartz Composer [8] and Open Frameworks [9]. VVVV was designed to facilitate the handling of large media environments with physical interfaces, real-time motion graphics, audio and video that can interact with many users simultaneously [7]. Quartz Composer is a node-based visual programming language provided as part of Xcode in Mac OS X [1]. Open Frameworks is a C++ library designed to assist the creative process by providing a simple and intuitive framework for experimentation [9].

Although the above tools are already convenient to use, there are still some obstacles that prevent media artists from easy creating their media contents using the tools [3]. ECAS was developed to be a cheap, easy to use, stable and efficient multi-platform media art authoring tool. The speed of ECAS is as fast as C/C++ based application thanks to the Java Native Interface and Just-in-time [10] compiling technology. Fig. 2(a) shows the case for testing face detection in ECAS version 1.0. The face detect function block is used for detecting faces inside one image frame. First, sequential images are grabbed from camera block then the camera block sends the data to the face detect block. The face detect block would detect human faces at each frame then send the result to the code box. Test code is set inside the code box block in order to retrieve the face information and draw it to the image buffer. Fig. 2(b) shows the same function implemented in *Processing*. Users need to write down about 50 lines code, but ECAS only uses five blocks.

The remainder of this paper is organized as follows. Section II discusses the whole architecture of ECAS and *Processing*,

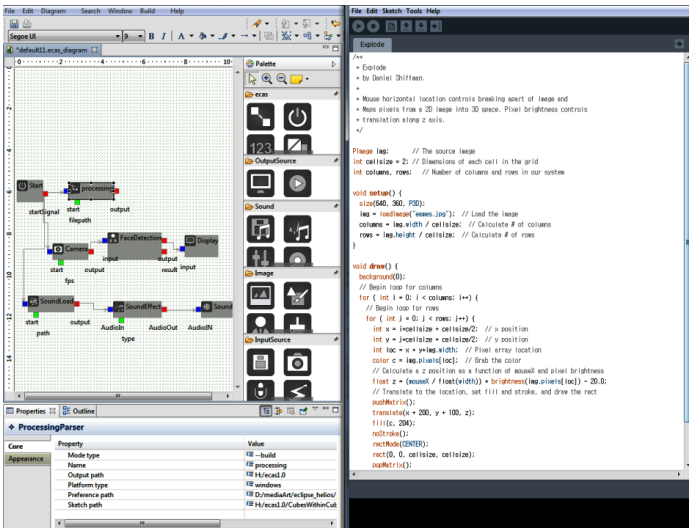
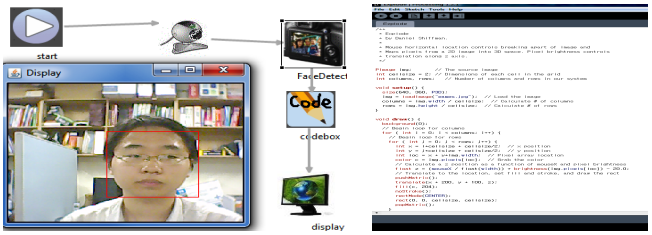


Fig. 1. ECAS and *Processing*



(a) Video test

(b) Face detection in *Processing*

Fig. 2. Face Detection test

respectively. Subsection II(A) introduces the main architecture of *Processing*. Subsection II(B) shows the architecture of ECAS in the current state and Subsection II(C) shows the architecture of ECAS in the future. The integration procedure will be introduced in subsection II(D). Section III gives the experimental result and conclusion.

II. Architecture

A. Architecture of Processing

The UI part of *Processing* is written in Swing [11], while the UI part of ECAS is written in SWT [12]. There is no big difference at performance between these two libraries. So we focus on the analysis to the logic part. Fig. 3 shows the main architecture of *Processing*. The initial input of *Processing* is text files whose extension is "pde". Then the source file would be translated to java files by the translator. The procedure is the most important part in *Processing*. The generated java files are directly complied to the executive class files by the Batch Compiler class which is defined inside JDK. Finally, JVM will run the Byte code to show the execution result.

As it is said before, the translation part is the key part in the whole procedure. There are totally two grammar files defined inside *Processing* project. One is "java15.g" and the other

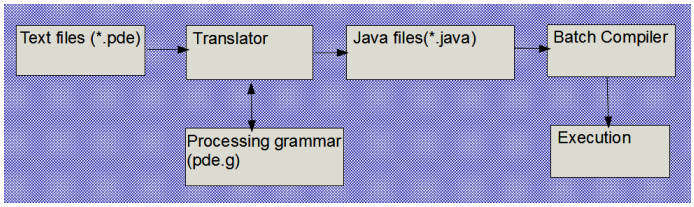


Fig. 3. *Processing* architecture

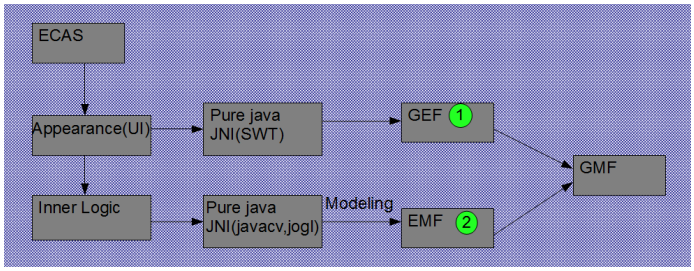


Fig. 4. ECAS architecture

is "pde.g". These two files are the grammar files from the ANTLR V2 [13], which is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators form grammatical descriptions containing actions in a variety of target languages. In the case of *Processing*, the target language is Java. "pde.g" is the grammar definition file of *Processing* language itself and "java15.g" is the grammar definition file of java language version 1.5. "pde.g" depends on "java15.g" when it translates Abstract Syntax Tree of *Processing* language to the Java language.

B. Architecture of ECAS in the current state

The ECAS system consists of two major parts: graphical user interface and program's logic. Usually, Graphical Editing Framework (GEF) [14] and Eclipse Modeling Framework (EMF) [15], [16] are used for designing GUI and program's logic, respectively. But Eclipse provides more convenient way to generate GUI and program's logic simultaneously. The framework used here is GMF which is the combination of GEF and EMF. That is, instead of designing each part separately, infrastructure of the whole program is firstly designed using GMF and then the generated source code would be further modified manually to be fully functional. Fig. 4 shows these relationships.

First step is to create domain model, actually this is the part which is very closely related to program's logic. In the current version of ECAS, about 42 classes are defined inside the domain model file (Fig. 5). The basic class here is named *Process*, because each block can be seen as a process when program is running. Seven methods are defined inside the *Process* class. They are listed below:

- void connect(*Process* target)
- void receiveMessage(Data data)
- void init() throws Exception
- void start()

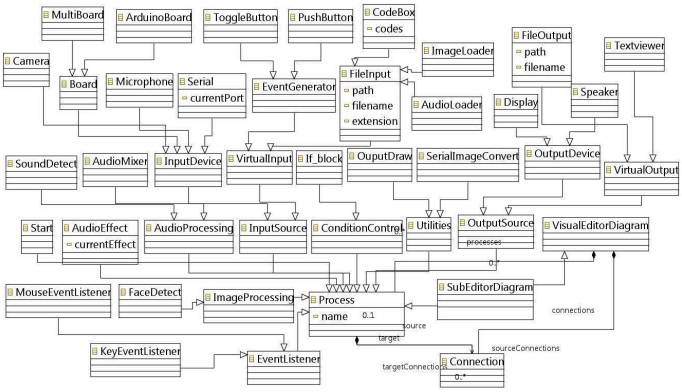


Fig. 5. ECAS function modeling

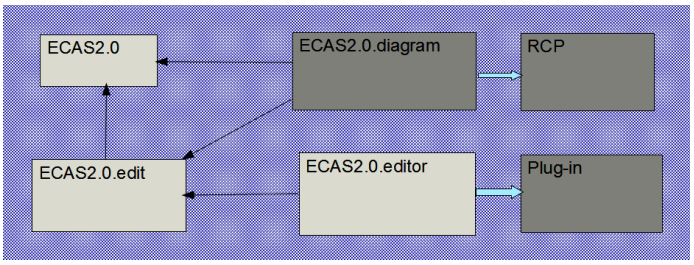


Fig. 6. ECAS structure

- void process(Data param)
- void destroy()
- void stop() throws InterruptedException

The default generated source code is just the skeleton of the whole program. It already can run in this state, but the program cannot do real functions what artists want. Additional functions should be extended manually. As it was mentioned before, there are two main parts: one is GUI and the other is logic. We focus on the logic part here. Fig. 7 shows the main concept of each function block. When the program is running, each node receives message from the previous node and stores data to the *input* variable. In order to further process, the node first calls *get*() method to get the data from the *input* then uses *process*() method to do real logical processing, finally, the node will call *send*() method to transfer the result to the next nodes.

C. Architecture of ECAS in the future

Every block used inside ECAS is implemented as a thread. Although the thread can go to sleep when there is nothing to do, it still remains inefficient compared to those using static compiling method. We will shortly describe the architecture of ECAS in the future.

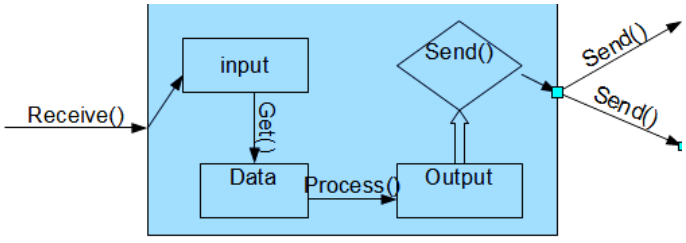


Fig. 7. process flow

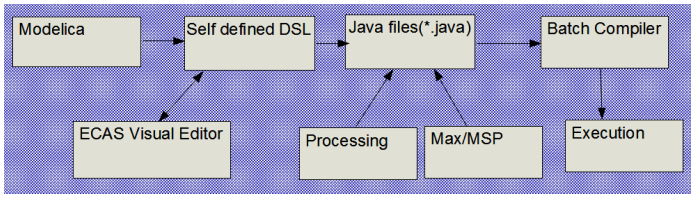


Fig. 8. ECAS logic architecture in the future

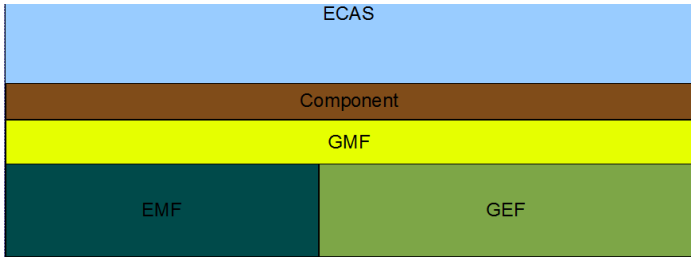


Fig. 9. ECAS UI architecture in the future

Fig. 8 shows the architecture from the logical perspective. Currently, visual information is stored inside a xml file. We will define a domain specific language which could describe the visual relationship of the nodes more efficiently. We are considering subset of Modelica language [17], which is an object-oriented, declarative, multi-domain modeling language for component oriented modeling of complex systems. We are developing language parser using ANTLR and Xtext [18].

Fig. 9 shows the architecture from the UI part. GMF has already provided us about 23 extension points. One could implement some functionality by extending one of these extension points. We would provide component in the future, which would implement the most of the basic extension points GMF provides. Then we would let our own extension points to the third party. This mechanism could provide much more flexibility to the current ECAS system.

D. Integrate Processing into ECAS

As it is shown in Fig. 10, we defined the five additional attributes inside ProcessingParser function block. They are Mode type, Output path, Platform type, Preference path and

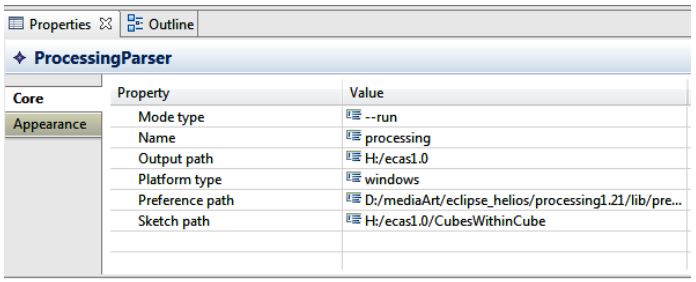


Fig. 10. Integrate *Processing* into ECAS

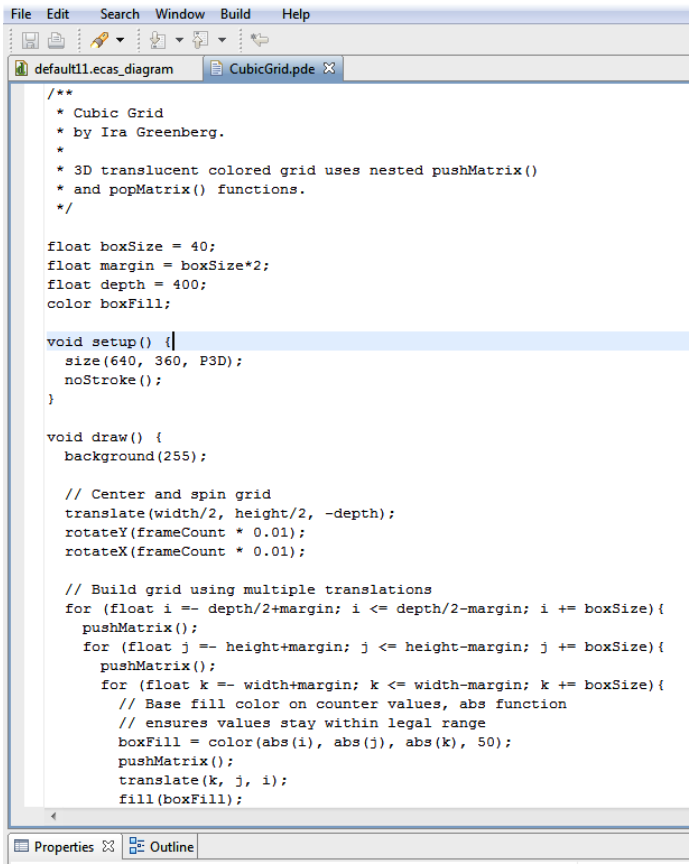


Fig. 11. Editing PDE files inside ECAS

Sketch path. There are seven modes inside *Processing*: ”–help”, ”–preprocess”, ”–build”, ”–run”, ”–present”, ”–export-applet”, ”–export-application”. The available platform types are ”windows”, ”linux”, ”macosx”. Users could modify properties using property sheet. One can also modify the corresponding pde source files by double clicking the Processing-Parser node as it is shown in Fig. 11. It is implemented by overriding **perfomRequest(Request request)** function inside the ProcessingParserEditPart.

III. Experiment and Conclusion

Fig. 12 shows the test result. We put camera display and ProcessingParser together in one diagram editor. The result is the same as we expected.

The procedure of integrating *Processing* into ECAS is presented in this paper. It can provide media artists more efficient and easier way of creating media so that they could create art contents by using existing *Processing* resources directly. Currently the *Processing* can only be converted to the java language version 1.5. In order to get higher compatibility, we need to rewrite java15.g grammar file so that the translation could support latest java version.

Acknowledgment

This work was supported by the Brain Korea 21 Project in 2010.

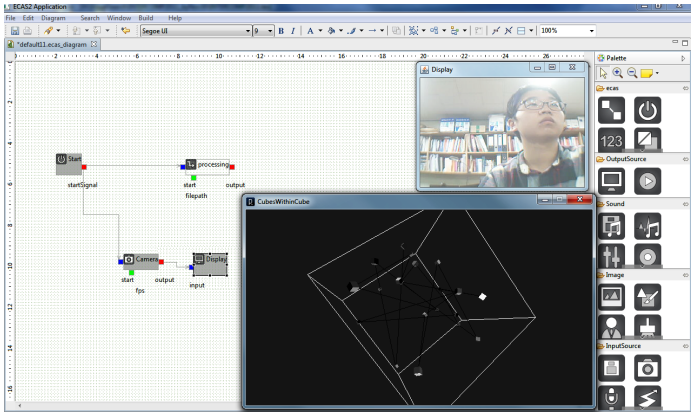


Fig. 12. Test result

References

[1] Wikipedia. http://en.wikipedia.org.
[2] Processing - open source programming language for media art. http://www.processing.org.
[3] Songlin Piao, Jae-Ho Kwak, and Whoi-Yul Kim. Research on eclipse based media art authoring tool for the media artist. In *Entertainment Computing - ICEC 2010*, volume 6243 of *Lecture Notes in Computer Science*, pages 342–349. Springer Berlin / Heidelberg, 2010.
[4] Graphical modeling framework. http://www.eclipse.org/modeling/gmf.
[5] Cycling74 - tools for media. http://www.cycling74.com.
[6] Puredata community site. http://puredata.info.
[7] A multipurpose toolkit. http://www.vvvv.org.
[8] working with quartz composer. http://developer.apple.com.
[9] open source c++ toolkit for creative coding. http://www.openframeworks.cc.
[10] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM Syst. J.*, 39(1):175–193, 2000.
[11] Bruce Eckel. *Thinking in Java*. Prentice Hall, Upper Saddle River, NJ, 4. edition, 2006.
[12] Matthew Scarpino, Stephen Holder, Stanford Ng, and Laurent Mihalkovic. *SWT/JFace in Action: GUI Design with Eclipse 3.0 (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.
[13] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.
[14] Eclipse graphical editing framework. http://www.eclipse.org/gef.
[15] Eclipse modeling framework. http://www.eclipse.org/modeling/emf.
[16] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009.
[17] Wladimir Schamai. Modelica modeling language (modelicaml): A uml profile for modelica. Technical report, 2009.
[18] Moritz Eysholdt and Johannes Rupprecht. Migrating a large modeling environment from xml/uml to xtext/gmf. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH ’10, pages 97–104, New York, NY, USA, 2010. ACM.