Algoritharium: Facilitating an Early Focus on Algorithms in an Objects-Early CS1 Course

Sridhar Narayan, Jack Tompkins, and Gene Tagliarini
Department of Computer Science
University of North Carolina Wilmington
Wilmington, NC 28403

Abstract— *Introducing large-scale problems early in the CS1 course has been shown to be an effective way to teach algorithmic concepts. Adopting this approach in a CS1 course taught in Java, however, presents some significant challenges. This paper describes a tool, the* Algoritharium, *that facilitates the process. The Algoritharium allows CS1 students to explore image processing algorithms by developing* plug-ins *for an image viewer. This approach allows students to circumvent many of the language-specific constraints, while enabling a focus on algorithmic issues and encouraging experimentation. A short series of exercises demonstrate that this approach can be adopted as early as the first day of the course and thus supports an early focus on algorithms in an objects-early CS1 course.*

Keywords

CS1, Java, image processing, algorithm, plug-in

1. Introduction

Introducing large-scale problems early in the CS1 course has been shown to be an effective way to teach algorithmic concepts [1], [2]. Computer graphics and image processing, in particular, affords numerous opportunities for students to encounter interesting and challenging algorithms and data structures, is visually appealing, and has the potential to mitigate the frustration that novice programmers often encounter - even an incorrectly implemented image processing algorithm can produce suprisingly pleasing results! The number and variety of projects[3], [4], [5] that incorporate computer graphics and image processing into a CS1 course is a measure of the popularity of this approach.

As an illustrative example of this approach, consider the following image processing project assigned to students in a CS1 class[6]. The project requires students to write a program that can produce a single-color 800x600 image in *binary* Portable Pixel Map (PPM) format[7], using only basic constructs of the C programming language. While not a commonly used file format, the PPM format has the advantage of being simple, with an ASCII header that includes a format code identifier, image dimensions, and a maximum intensity value, followed by image data with each pixel represented by three bytes corresponding to the red, green, and blue intensities. The C program for this assignment requires a *main* function, *printf*, and a *for* loop - constructs that are well within the capacity of beginning students - and is thus quite tractable for a freshman in a CS1 class. Note that this exercise requires students to use software like *Irfanview* [8] to display the resulting image.

However, adopting this approach in an object-oriented CS1 course taught in Java can be a daunting prospect. The constraints of the Java programming language, when combined with the desire for an early focus on objects, can present some significant challenges to the instructor and students alike. For instance, the exercise outlined above would require, at a minimum, at least a cursory understanding of the concepts of class, method, and exceptions, keywords such as *public*, *static* and *void*, and the ability to use elements of the Java I/O mechanism like *DataOutputStream* and *FileOutputStream*.

Two approaches are commonly adopted to help students surmount the steep learning curve presented by Java. One approach involves presenting all the necessary language constructs in a superficial manner and asking students to employ the constructs without particularly understanding them. This approach is distracting at best, and potentially very dangerous to a student's understanding. Beginning students can come to regard programming as a priesthood where ill-understood incantations lead, inexplicably, to auspicious outcomes. A second approach involves the use of APIs to hide complexity[3], [9], [1]. However, determining the proper level of detail in the API can be difficult. APIs that require students to understand implementation details can overwhelm the beginning student. On the other hand, APIs that hide most of the complexity, can preclude many learning opportunities and make programming seem like 'magic' to the beginner. Worse, it can predispose some students to think that the essence of programming is unearthing the right API on Google.

This paper describes a tool, the *Algoritharium*, that facilitates an early focus on algorithms in a CS1 course taught using Java. Using a process that resembles the development of *plug-ins*, the Algoritharium uses a *minimal* API to allow CS1 students to explore image processing algorithms by developing *extensions* for an image viewer. Thus, the Algoritharium allows students to gain valuable experience in developing interesting and challenging algorithms while *using* objects. A series of short exercises demonstrate that this approach can be used as early as the first day of the

course and thus supports an early focus on algorithms in an objects-early course.

2. Algoritharium

As the name suggests, the Algoritharium is envisioned as an environment in which students can explore algorithms. In its current form, the focus of the project is on image processing algorithms. It consists of the *ImageViewer* class and the *Image* interface, packaged into an executable JAR file. Using the Algoritharium is a two-step process:

1) The first step involves launching an instance of ImageViewer by executing the JAR file. The image viewer provides the following functionality. As shown in Figure 1, the *File* menu option in the image viewer invokes a file browser that allows images to be opened, displayed, and saved. A *File-New* menu option creates and displays a 200x150 image in a default color. The *File-Duplicate* menu option can be used to launch multiple image viewers and supports image processing operations that involve multiple images. The ImageViewer supports several common image formats, including *.jpg, .gif, .png, and .ppm.*
2) In the second step, students develop a class with methods that implement image processing algorithms, using a development environment of their choice.
   a) As shown in Figure 2, the *Code-Load* menu option allows students to *load* their compiled code into the image viewer. Upon recompilation, the modified class file can be *dynamically* reloaded into the image viewer using the *Code-Reload* option.
   b) Methods contained in the student's class file are automatically added as *sub-options* to the *MyOps* menu option, as shown in Figure 3. These methods can then be executed in the context of the image displayed in the viewer. Figure 3 shows the image after the student-developed method *redFilter* has been applied to it. As shown in Figure 4, the *Algoritharium* presents error messages generated during the execution of student-written algorithms in a modal dialog box, selectively emphasizing the references to student-developed code in the process.
   c) The image currently displayed in an image viewer can be changed at any time. The student-developed methods currently loaded into the image viewer can then be used with the new image.

The following sections illustrate the classroom use of the Algoritharium in a series of exercises that can potentially be assigned as early as the first class.

2.1 Exercise 1: Modify a single pixel in an image

Students are asked to write a Java program that can change the color of one pixel in the current image to red. As preparation for this exercise, the class is provided with the following background. An image is a collection of pixels. The *getImage* method can be used to retrieve an image from an image viewer. The color of a pixel can be modified by the *setPixelColor* method by specifying the location of the pixel and its new color. A typical student response is shown below:

Fig. 1
IMAGE VIEWER MENU OPTIONS

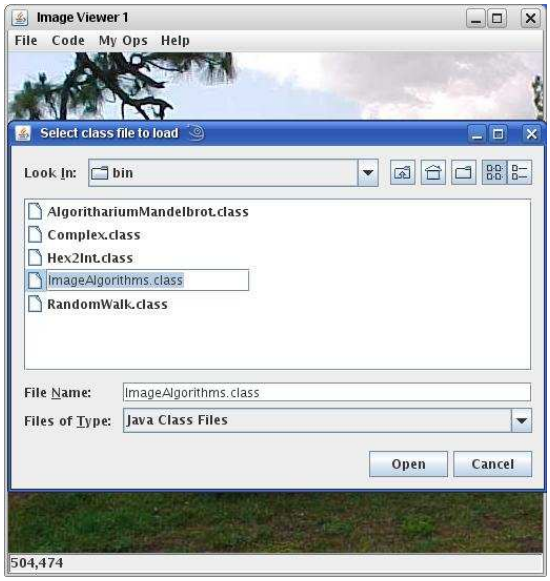

Fig. 2
LOADING STUDENT DEVELOPED CODE INTO AN IMAGE VIEWER

```
import algoritharium.*;
import java.awt.Color;
public class StudentImageAlgorithms
{
  public void oneRedDot()
```

Fig. 3
EXECUTING STUDENT DEVELOPED CODE IN AN IMAGE VIEWER

Fig. 4
FILTERED ERROR MESSAGES APPEAR IN A MODAL DIALOG BOX

```
    {
        Image img = ImageViewer.getImage();
        img.setPixelColor(20,20,Color.RED);
    }
}
```

The student's code functions as a *plug-in* and the corresponding class file can be loaded into the image viewer using the *Code-Load* menu option. The *oneRedDot* method becomes available as a new option under *MyOps*. Selecting this option causes the *oneRedDot* method to be executed in the context of the image displayed in the image viewer.

Note that since the student's code is plugged-in to the image viewer tool, this approach does not require a *main* method in the student-developed class. Thus, a discussion of *public static void main(String [] args)* can be deferred, focusing the student's attention instead on their algorithm for manipulating the image. However, the method still introduces classes and objects while emphasizing the role of message passing in an object-oriented environment, and allows parameters to be introduced.

2.2 Exercise 2: Modify several adjacent pixels in one row of the image

This is a natural follow-up to exercise 1. Students respond by *adding* a method named *manyRedDots* to the class described above. The modified class file can be loaded into the image viewer and *both* methods, *oneRedDot* and *manyRedDots*, become available as options under *MyOps*. A typical student response uses several statements, each of which modifies the color of a single pixel. This exercise demonstrates the important concept of a computer program as a *sequence* of instructions. It also prompts the students to observe that computer programs often perform repetitive tasks and naturally lays the groundwork for introducing the concept of *iteration* next.

2.3 Exercise 3: Use *iteration* to modify several adjacent pixels in one row of the image

The redundancy in *manyRedDots* makes the need for iteration obvious. A discussion of *for* loops leads to a refinement of *manyRedDots*, in which the sequence of statements is replaced by a single statement executed inside a *for* loop.

2.4 Exercise 4: Make the entire image red

Having changed the color of several pixels, a natural next step is to write a method that can change the color of every pixel in the image. This exercise introduces nested *for* loops.

2.5 Exercise 5: Red/green checkerboard

This exercise builds on the previous exercise and asks students to write a method that can be used to produce a red and green checkerboard pattern. It naturally adds *selection* to the student's programming repertoire and for the first time requires the development of a somewhat non-trivial algorithm. It also provides an opportunity for students to develop different algorithms to accomplish the same task. For instance, one implementation may redundantly include code to toggle the current pixel color in two places. This provides an opportunity to introduce the notion of *delegating* responsibility and allows students to develop a separate helper method for toggling the color. An alternative implementation in which the red and green colors are stored in a one-dimensional array may be used to introduce *arrays*.

This exercise also provides students with the opportunity to be creative and explore the creation of other patterns besides the one assigned. For instance, the authors have successfully used this exercise as a precursor to a so-called *Flags* project in which students create flag-like images that incorporate various geometric patterns.

2.6 Exercise 6: Implement a color filter

The previous exercises do not require the student to consider the current state of a pixel before manipulating it. Implementing a color filter requires students to understand that an object encapsulates properties and behavior. Specifically, that the color of a pixel has red, green, and blue components that can be accessed and manipulated independently. It also gives students the opportunity to reflect upon the data management required even for a small problem - a 640x480 image has over 300,000 pixels, each of which has 3 color components for a total of close to a million pieces of data.

The six exercises presented above demonstrate how the Algoritharium can be used in an objects-early course to provide an early emphasis on algorithms in the context of an engaging activity like image processing. Note that the exercises require only a minimal familiarity with the Algoritharium API - the *ImageViewer.getImage* method and the *getPixelColor* and *setPixelColor* methods in the *Image* interface. Thus the API does not distract the students from the core concepts of objects, message passing, and program control structures.

The API methods employed above have been used by students in the author's classes to implement a rich set of image processing algorithms including, for instance, gray scaling, adding scan lines, edge detection, object detection, and image composition. For operations that require more support, the Algoritharium API provides a more complete set of methods that can be used to introduce, among other topics, one and two-dimensional arrays. A complete listing of the API for the Algoritharium appears in the appendix.

2.7 Image algorithms involving multiple images

Many image processing algorithms involve two or more images[10]. For instance, a color transfer algorithm[11] computes the color characteristics of one image and applies it to a second. The Algoritharium supports such computations by allowing a second image viewer to be instantiated using the *File-Duplicate* menu option. Both image viewers can be used concurrently, and each viewer is identified by a numeric id that is visible in the image viewer. Different images can be opened in the two image viewers and each image can be accessed by referencing its viewer identifier as shown in the example below.

```
public void twoImageOperation() {
 Image source = ImageViewer.getImage(1);
 Image target = ImageViewer.getImage(2);

 //Code for image manipulation
}
```

Operations involving concurrent access to multiple images, for instance compositing two images to produce a third one, can be implemented by instantiating more image viewers.

2.8 Supporting user interaction

The ImageViewer also provides feedback to the user via mouse interaction. Clicking on an image reveals the (x,y) coordinates of the point. This information can be used to identify a region of an image, for instance when implementing a collage maker. Student-developed plug-ins can also interact with the user via a console.

2.9 Support for animation

Student developed methods are executed in an independent thread, separate from the event handling thread used by the Java runtime. This allows students to animate their image processing algorithms by introducing small delays into their code by using the *ImageViewer.delay()* method. For instance, this is a particularly effective way for students to see that reordering the loop indices in a doubly-nested *for* loop used to paint all pixels in an image causes the pixels in the image to be painted either by rows or by columns.

3. Results

The Algoritharium tool has been introduced in two sections of the CS1 course at the authors' institution. Preliminary results indicate that students feel empowered by the process of writing plug-ins to enhance the image viewer, an experience equivalent to that of customizing an Adobe Photoshop®-like application. The ability to install their code with a single click, and within seconds of writing the code see the effect on various images adds excitement and a sense of accomplishment. Students also derived a sense of satisfaction by being able to execute their code from a graphical user interface, - a small accomplishment that nevertheless helps bridge the gap between their modest initial programming efforts and what they perceive as "real" software. Learning that the image viewer tool is implemented in Java also gave the students a sense for what they can accomplish as their programming skills mature.

Students enrolled in the two sections that used the *Algoritharium* were asked to complete a general survey that assessed the students' perception of their understanding of key topics covered in the course. The general survey was also administered to a third section of the CS1 course in the same semester at the author's institution that did not use the *Algoritharium*, and served as a control. Figure 5 compares the average responses of the students in the

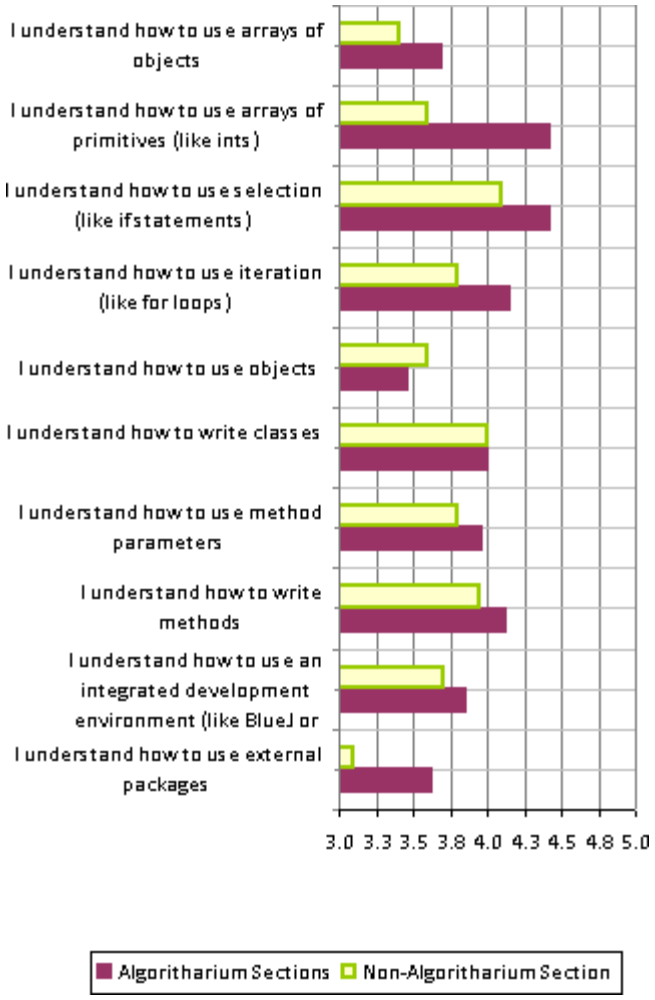

Fig. 5
COMPARISON OF STUDENTS RESPONSES IN THE ALGORITHARIUM SECTIONS TO THE CONTROL SECTION

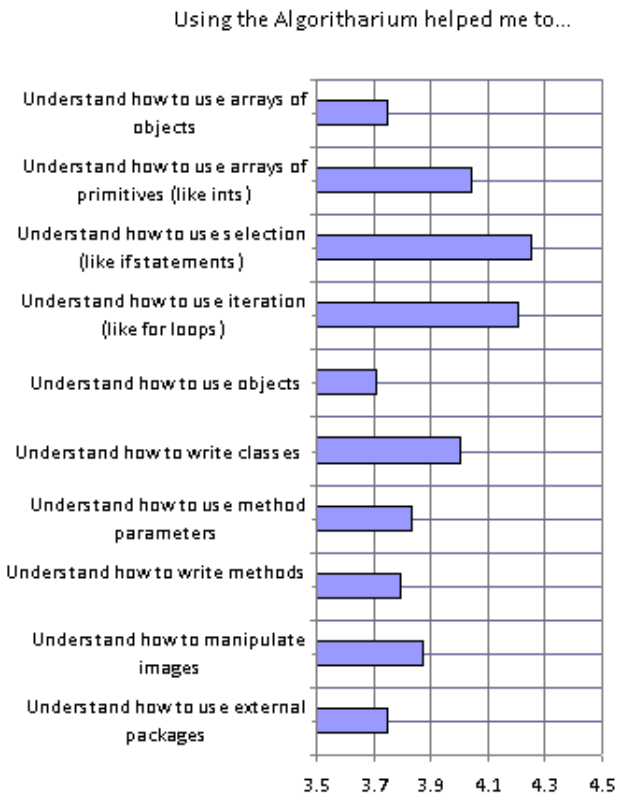

Fig. 6
STUDENT RESPONSES TO ALGORITHARIUM SPECIFIC QUESTIONS

sections using the Algoritharium, to the responses of the students in the section that did not use the Algoritharium. The biggest difference in the student responses appears in their perception of their understanding of iteration and selection program control structures, and array processing. The difference in the responses to the question about arrays of primitives is statistically significant at the $\alpha = 0.01$ level. These results are consistent with the explicit early focus on algorithms and image processing in the sections using the *Algoritharium*.

Figure 6 shows average student responses to a second survey administered only to students in the two sections that used the *Algoritharium*. As with the first survey, the strongest responses appear in questions related to the students' perception of their facility with iteration and selection program control structures, and array processing. Taken together, the survey results suggest that the *Algoritharium* has a strong, positive impact on student understanding of the programming constructs that are central to algorithm development in a CS1course.

4. Conclusions

Presenting an engaging activity like image processing early in a CS1 class is a desirable goal. Image processing exercises have visual appeal and present numerous opportunities for challenging algorithms. However, the syntactical and organizational constraints of an object-oriented language like Java can be an impediment to the early introduction of image processing in a CS1 class. This paper describes a tool, the *Algoritharium*, that mitigates these difficulties. Using a process that resembles the development of *plug-ins*, the Algoritharium allows CS1 students to explore interesting and practical image processing algorithms by developing *extensions* for an image viewer. Since the image viewer can be executed from a JAR file and provides mechanisms for file I/O, beginning students are not distracted by Java artifacts such as exceptions and OutputStreams. At the discretion of the instructor, even the obligatory discussion of *public static void main(String [] args)* can be deferred. Instead, students can focus on the task of algorithm development.

As illustrated by the six exercises, the Algoritharium can be used to introduce beginning students to algorithm development in an objects-early course in a carefully scaffolded

sequence. The exercises require only a minimal familiarity with the Algoritharium API and therefore the API does not distract the students from the core concepts of objects, message passing, and program control structures. At the same time, the API presents a sufficiently detailed view of image processing to afford students the opportunity to implement interesting and challenging algorithms.

The approach presented in this paper, while supporting early *use* of objects, favors a focus on algorithms. For instance, the class developed by students is primarily a utility class that serves as a container for methods. This approach tacitly defers the presentation of a class as a template with properties and behaviors. This reflects our belief that gaining confidence *early* in developing algorithms using objects is good preparation for a *later* discussion of the design of objects.

5. Acknowledgments

This work was supported in part by the National Science Foundation under award number 0722313, *CPATH EAE: τεχνη - Evaluation, Adoption and Extension*.

References

[1] R. Wicentowski and T. Newhall, "Using image processing projects to teach CS1 topics," *SIGCSE Bull.*, vol. 37, no. 1, pp. 287–291, 2005.

[2] G. Shultz, "Integrating 3d graphics into early CS courses," *J. Comput. Small Coll.*, vol. 21, no. 3, pp. 169–178, 2006.

[3] B. Stephenson and C. Taube-Schock, "Quickdraw: bringing graphics into first year," in *SIGCSE '09: Proceedings of the 40th ACM technical symposium on Computer science education*. New York, NY, USA: ACM, 2009, pp. 211–215.

[4] S. Matzko and T. Davis, "Using graphics research to teach freshman computer science," in *SIGGRAPH '06: ACM SIGGRAPH 2006 Educators program*. New York, NY, USA: ACM, 2006, p. 9.

[5] T. Davis, R. Geist, S. Matzko, and J. Westall, "texnh: a first step," in *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*. New York, NY, USA: ACM, 2004, pp. 125–129.

[6] S. Matzko and T. A. Davis, "Teaching CS1 with graphics and C," *SIGCSE Bull.*, vol. 38, no. 3, pp. 168–172, 2006.

[7] http://netpbm.sourceforge.net/doc/ppm.html.

[8] http://www.irfanview.com.

[9] K. Hunt, "Using image processing to teach CS1 and CS2," *SIGCSE Bull.*, vol. 35, no. 4, pp. 86–89, 2003.

[10] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1992.

[11] E. Reinhard, M. Ashikhmin, B. Gooch, and P. Shirley, "Color transfer between images," *IEEE Computer Graphics and Applications*, vol. 21, no. 5, pp. 34–41, 2001.

6. Appendices

6.1 The ImageViewer API

static Image getImage() Returns the image associated with the current ImageViewer instance

static Image getImage(int number) Returns the image associated with the ImageViewer instance identified by the parameter *number*

int getViewerId() Return the numeric identifier for the current ImageViewer instance

static void createImage(Color [][] c) Creates an image corresponding to the parameter *c* in the current ImageViewer instance

static void createImage(int number, Color [][] c) Creates an image corresponding to the parameter *c* in the ImageViewer instance identified by the parameter *number*

static void setImage(int number, Image image) Associate the image referenced by the parameter *image* with the ImageViewer instance identified by the parameter *number*

static void delay(long interval) Pause method execution for *interval* milliseconds

6.2 The Image API

Color[] getColumn(int col) Returns a one-dimensional array of Color objects corresponding to the pixels in the column specified by the parameter *col*

int getHeight() Returns the height (in pixels) of this image

Color getPixelColor(int x, int y) Returns the color of the pixel at the location specified by the parameters *x* and *y*

Color[][] getPixelColors() Returns a two-dimensional array of Color objects corresponding to the colors of the pixels in the image. The array has *imageHeight* number of rows, and *imageWidth* number of columns.

Color[][] getRectangle(int x, int y, int w, int h) Returns a two-dimensional array of Color objects corresponding to the pixels in the rectangular area described by the parameters

Color[] getRow(int row) Returns a one-dimensional array of Color objects corresponding to the pixels in the row specified by the parameter *row*

int getWidth() Returns the width (in pixels) of this image

void setPixelColor(int x, int y, Color c) Sets the color of the pixel at the location specified by the parameters *x* and *y* to the color specified by the parameter *c*

void setPixelColors(Color[][] c) Updates the pixels in the image to correspond to the colors in the array specified by the parameter *c*

void setRectangle(int x, int y, Color[][] c) Paints a two-dimensional array of Color objects at the coordinates given in the parameters

void setRow(int row, Color[] c) Updates the pixels in the row specified by parameter *row* to correspond to the colors specified in the one-dimensional array *c*