Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 178 (2007) 33–41
www.elsevier.com/locate/entcs

JHAVÉ – More Visualizers (and Visualizations) Needed

Thomas L. Naps[1]

Computer Science Department
University of Wisconsin Oshkosh
Oshkosh, WI, USA

Guido Rößling[2]

Computer Science Department
Technische Universität Darmstadt
Darmstadt, Germany

Abstract

We recap the results of an ITiCSE 2002 Working Group report [4], which set the stage for the work described here. That work has resulted in the newest release of a system called JHAVÉ, which fosters active engagement on the part of learners by providing a set of standard support tools for certain types of Algorithm Visualization (AV) systems. These *AV engines* must implement the JHAVÉ *Visualizer* interface. In return, such engines have convenient access to the engagement-based tools offered by JHAVÉ. The details of adapting one such engine, ANIMAL, are also described.

Keywords: Algorithm visualization, tree animations, interaction, engagement

1 Background

The report of an ITiCSE 2002 Working Group codified an *engagement taxonomy* for the modes for allowing students to be *active participants* in exploring an algorithm with an AV system [4]. The report defined four active categories of engagement:

- Responding,
- Changing,
- Constructing,

[1] Email: naps@uwosh.edu
[2] Email: roessling@acm.org

doi:10.1016/j.entcs.2007.01.035

- and Presenting.

Students *responding* to questions regarding the current visualization are usually asked to predict the algorithm's behavior ("what will happen next?"), or to reflect on more conceptual aspects ("which part of the code caused this effect?"). The AV system here becomes a supporting resource for posing questions and helping the student answer them.

Changing the visualization usually requires students to specify input that will cause a certain behavior. For example, the system could ask for an array that will cause exactly one value to end in its correct position during each loop iteration of Bubble Sort. After specifying the input set, the AV system can show the student if the input achieved the desired goal.

Constructing a personal visualization clearly entails a larger time commitment on the part of the student. A common technique for constructing a visualization is to first code it and then annotate it with calls to have graphics produced at "interesting events" during the algorithm's execution. Well-designed class libraries can make this relatively painless for the student to do. For example, a data structure object such as a graph may have a "display" method that can be called to produce an aesthetically pleasing picture of the object in the AV system without requiring the student to do much beyond a method call [2].

It is important to note that constructing the visualization does not always entail having the student code the algorithm. Systems such ANIMAL [5] and *ALVIS* [1] provide visualization designers with a collection of modeling tools that are particularly well-suited to algorithm visualization. In effect, they allow the user to create a movie about the algorithm working strictly from a conceptual perspective, without ever having to code the algorithm.

In the *presenting* category, the student is asked to use a visualization to help explain an algorithm to an audience for feedback and discussion. The visualization may or may not have been created by the presenter.

2 JHAVÉ – A Pedagogical Support Environment for AV Systems

Clearly, graphics alone are not sufficient to *effectively* deliver educational applications of algorithm visualization. Unfortunately, the extra tools that are needed for such effective deployment, namely "hooks" by which to actively engage the student with the visualization, often require more effort to produce than the graphic displays themselves.

This section of the paper describes JHAVÉ (Java-Hosted Algorithm Visualization Environment) [3], a system designed to overcome this impediment. JHAVÉ is not an AV system itself, but rather a support environment for AV systems (called *AV engines* by JHAVÉ). JHAVÉ aims to be a common platform for staging a set of different AV systems, enabling the integration of available tools into one smooth front-end, while also offering added value for each integrated AV engine.

In broad terms, JHAVÉ gives such an engine a drawing context on which it can render its pictures in any way that it wants. In return, JHAVÉ provides the engine with effortless ways to synchronize its graphical displays with:

- A standard set of *VCR-like controls*. These allow students to step through the visual display of the algorithm. Hence, there is a standard "look and feel" to the GUI used by the student that is essentially independent of the AV engine being used.
- *Information* and *pseudo-code windows*. These are essentially HTML windows where visualization designers can author static or dynamically generated content to help explain the significance of the graphical rendering of the algorithm. The information window is used for higher-level conceptual explanations. The pseudo-code window may display a pseudo-code description of the algorithm complete with highlighting of lines that are particularly relevant for the picture currently being displayed.
- *"Stop-and-think" questions*. The visualization designer can designate questions in a variety of formats – true-false, fill-in-the-blank, multiple-choice, and multiple-selection (multiple-choice with more than one right answer) – to "pop up" at key stages of the algorithm. The question will typically ask the student to predict what they will see next, facilitating the *responding* category of engagement described in the previous section.
- *Input generators*. These objects gather input data from a student, if the visualization supports the *changing* category described in the previous section. This input data can be fed to the visualization, allowing the student to learn whether or not their data set drives the pictures in the anticipated fashion.
- Meaningful *content generation* tools. These include a variety of class libraries to help an AV designer (who may be a teacher or a student) create a visualization – both its graphical display and the interaction support tools that JHAVÉ offers.

The AV engine must produce the visualizations of an algorithm by parsing a textual visualization script and then rendering its pictures based on the script. Each AV engine is free to define its own scripting language – indeed, this is what allows a great deal of variation among the engines. The algorithm to be visualized has to generate an appropriate script file, which is then parsed and rendered by the AV engine.

The reasons for this approach are tied to the client-server architecture of JHAVÉ. The server application manages the available algorithms and generates the visualization scripts that the client can display. In a standard session, a student first launches an instance of the client application, which displays a listing of available algorithms, tailored to the particular subject that the viewer is studying. The AV engines are included in the client distribution.

When the user selects an algorithm from that list, the client sends a request to the server, which will run a program that generates the script for that algorithm and sends the URL of the script file to the client. If the algorithm requires input from the user, the server sends an input generator object to the client, which opens

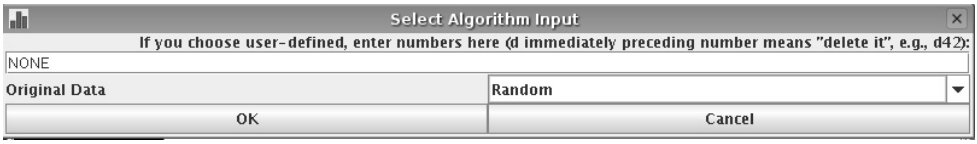

Fig. 1. Input generator gets data from user to direct the visualization

a simple frame with appropriate input areas (see Figure 1). Once the user fills out these areas, the client returns the user's input to the server as a data set to use when running the algorithm. After the script is generated on the server, the client downloads, parses, and renders it. The steps in this process are illustrated in Figure 2. The rendering is staged using JHAVÉ's VCR-like viewing controls, "stop-and-think" questions, and information/pseudo-code windows (Figure 3). The script generation program is usually written by a visualizer or educator, not by the end-user.

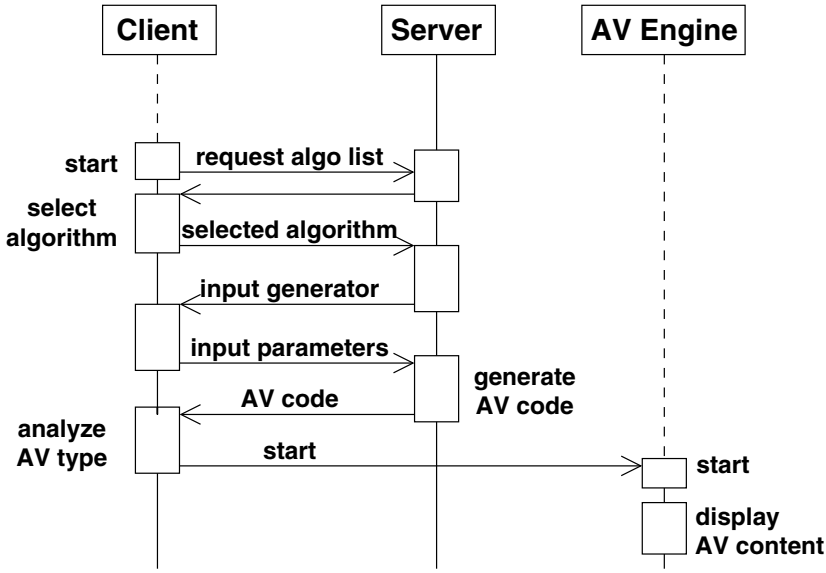

Fig. 2. Sequence of steps used to generate and display AV content

AV engines that presently "plug into" the newest version of JHAVÉ as clients in the fashion described above are Gaigs-ML (an XML-based data structure description language described in another PVW 2006 submission by McNally and Naps) and, more recently, ANIMAL and ANIMALSCRIPT [5]. To become a JHAVÉ AV engine, a renderer must declare that it is a subclass of JHAVÉ's abstract *Visualizer* class. As a consequence of this, it then must:

- Inform JHAVÉ of its capabilities by calling the *setCapabilities* method. JHAVÉ allows a Visualizer to support a subset of the following capabilities:
  - Stepping forward to the *next frame* in the visualization,
  - Stepping backward to a *prior frame* in the animation,
  - Going directly to a *particular frame* in the animation, skipping the rendering of all frames between that frame and the current frame,

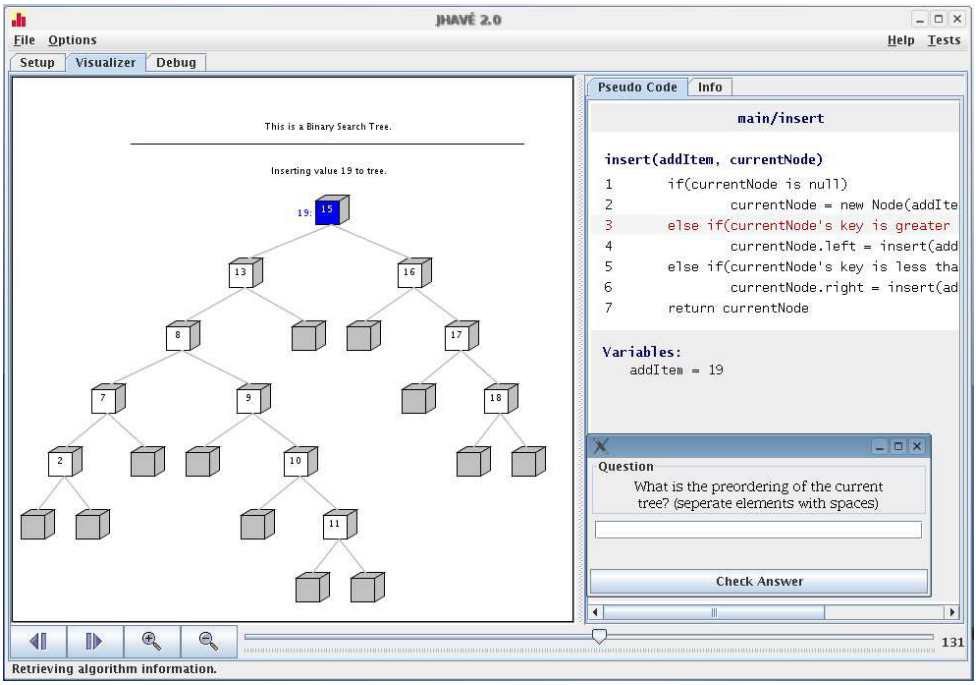

Fig. 3. AV engine renders picture in leftmost pane, with support from the JHAVÉ environment in the form of VCR-like controls, pseudo-code window, and stop-and-think question

- *"Zooming"* in or out on the picture in the current visualization frame,
- *Animation* mode. A Visualizer object that supports this mode is capable of delivering its visualization as a smoothly animated motion picture instead of as a slide show consisting of discrete snapshots. Slide show mode is the default unless a visualizer declares otherwise. Gaigs-ML operates in slide show mode, whereas ANIMAL and ANIMALSCRIPT offer richer animation capabilities.

JHAVÉ uses this information about the Visualizer's capabilities to display the appropriate subset of VCR controls in its GUI for that Visualizer. For example, Figure 3 shows that the controls for the Gaigs-ML Visualizer allow stepping forward, stepping backward, zooming, and direct access to a particular frame.

- Implement a constructor that receives an input stream as its only parameter. When JHAVÉ instantiates a particular Visualizer object, it calls this constructor, passing in the animation script, which exists as a URL on the server, for the input stream parameter.
- Implement three additional abstract methods from the base Visualizer class: *getCurrentFrame* returns the index number of the current frame in the visualization, *getFrameCount* returns the total number of frames in the visualization, and *getRenderPane* returns the Java component in which the engine renders the visualization.
- For each capability that it declares itself capable of supporting, it must override a method in the base Visualizer class to ensure the appropriate action is taken when the viewer uses the JHAVÉ GUI to trigger the event associated with that

capability.

If an AV engine encounters a tag for a question or a pseudo-code/documentation window during parsing, it can invoke a *fireQuestionEvent* or *fireDocumentationEvent* method in the Visualizer base class. These methods handle everything connected with processing that event and hence free the particular AV engine from concerning itself about the details of parsing and displaying the information associated with them. Instead that is all handled by the Visualizer base class. The AV engine itself can thus focus on rendering the visualization in its pane.

3 Integrating Animal and AnimalScript into JHAVÉ

Animal with its scripting notation AnimalScript [5] was an almost obvious candidate for integrating an "external" AV engine into JHAVÉ. Based on past cooperations and the fact that Animal already supported most controls, integrating Animal into JHAVÉ was expected to be easy.

Animal as integrated in JHAVÉ supports two formats - the internal, more efficient but somewhat less expressive and less readable format used by Animal, and AnimalScript. As mentioned above, the constructor of the visualization engine receives only the input stream. This means that the detection of the type of the underlying file should be performed before the constructor is actually invoked, as there may be no convenient way to close and reopen the input stream if the initial attempt at parsing it in the "wrong" format failed.

Therefore, the adaptation of Animal was spread over two classes. One class is responsible for handling Animal's internal format (compressed or uncompressed), while the other handles compressed or uncompressed AnimalScript. The code is mostly identical where JHAVÉ is concerned. In fact, most methods for mapping JHAVÉ's calls to the underlying Animal visualization engine contain between one and four lines of code.

While most of the work was done quickly, the main part of the work lay in preparing Animal for cooperation with JHAVÉ. Animal's animation window used to be a stand-alone frame which incorporates a rendering canvas and two extensive toolbars for controlling the animation progress, as shown in Figure 4.

As JHAVÉ has its own set of controls, only Animal's animation canvas was to be used. Due to interconnections between canvas and controls, the underlying Animal system had to be adapted somewhat. The net effect of this was that if an Animal or AnimalScript visualization is required, the tool will now start Animal, initialize all components, and return the animation canvas as the rendering area for JHAVÉ.

Another aspect which is probably specific to Animal concerned the numbering of steps. Both the JHAVÉ and the Animal GUI lets users jump directly to a step with a given number, assuming that steps are numbered consecutively and without gaps. When animations are generated in Animal's GUI front-end, each animation step receives a unique number. While these numbers are provided sequentially, the animation steps may be placed in an arbitrary order. Therefore, animations do not always start with step 1 – in fact, there may not be a step with the number 1!

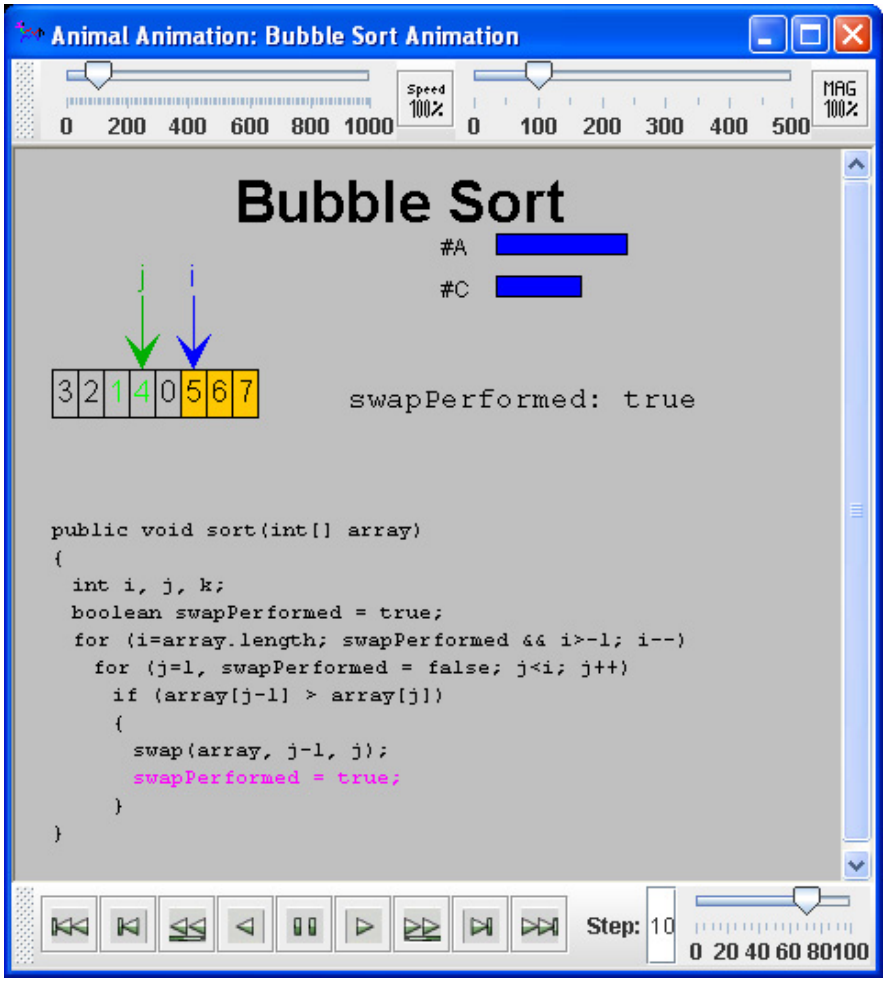

Fig. 4. A Bubble Sort animation running in ANIMAL in stand-alone mode

Additionally, the largest step number does not necessarily belong to the last step, and may also not be connected to the actual number of steps. Therefore, some additional code was written to ensure that when the user requests frame 3, he or she will see the content of the third step from the animation's start - no matter what the ANIMAL-internal step number of this step may be.

Once these adaptations were put into place, both ANIMAL and ANIMALSCRIPT could be integrated into JHAVÉ as new visualizers. We expect that adding other systems to JHAVÉ should be similarly easy.

Figure 5 shows an example animation window of JHAVÉ running ANIMAL as the visualization engine. At first glance, there is not much difference visible in the GUI provided by JHAVÉ between this Figure and Figure 3. ANIMAL adds an animation and pause button, compared to the visualization engine presented in Figure 3, as ANIMAL supports smooth animations. The pseudo code / information area to the right is currently empty, as we are still working on getting content for this area embedded into the existing ANIMALSCRIPT animation files.

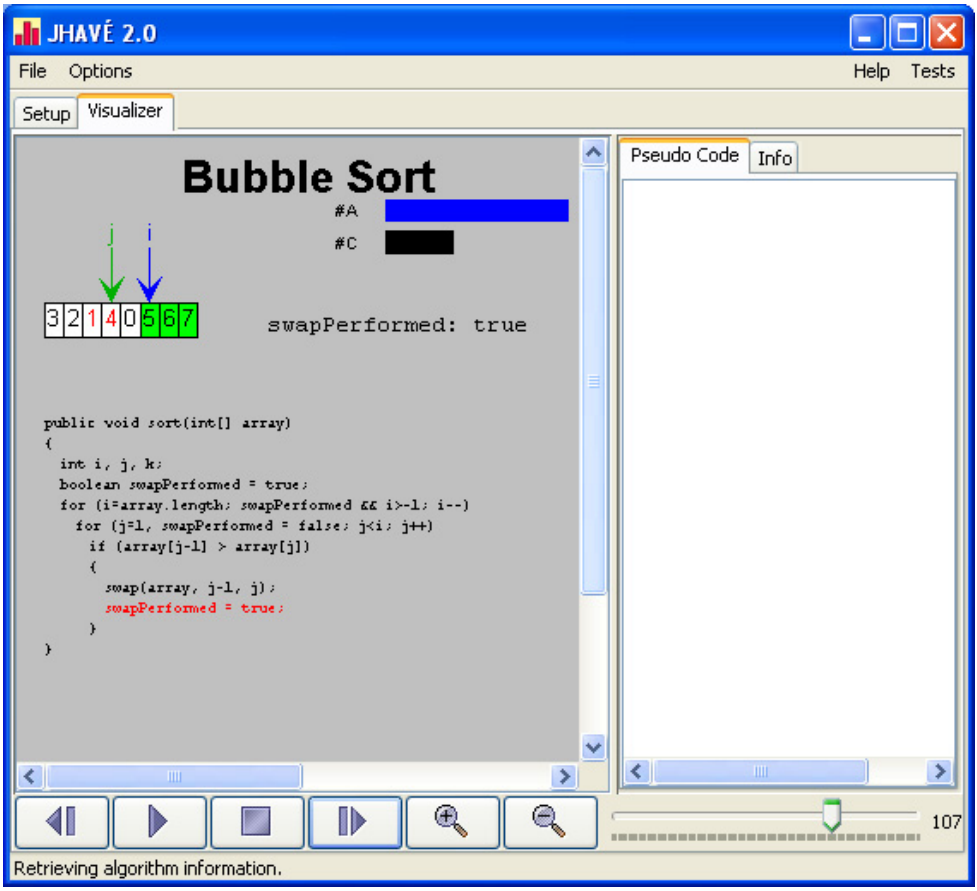

Fig. 5. An Animal animation running in JHAVÉ

Additionally, there are only small differences between the view shown in Figure 5 and Figure 4. Mainly the Animal-specific toolbars for speed, magnification, and step control shown in Figure 4 are replaced by JHAVÉ's control elements, plus the – currently still empty – documentation and pseudocode frame. The visible differences in the display quality between the two Figures are a result of different scaling factors employed, due to the differences in frame width between Animal's animation window and the full JHAVÉ window.

4 Conclusion and Future Directions

It is our sincere hope that the work presented here will encourage other AV developers who use a scripting language in their work to consider providing a version of their system as a JHAVÉ AV engine satisfying the Visualizer interface that we have described. By doing so, they will help make available to students an increasing base of interesting visualizations that are conveniently instrumented with the learning devices we have described here.

References

[1] Hundhausen, C. D. and S. Douglas, *SALSA and ALVIS: A Language and System for Constructing and Presenting Low Fidelity Algorithm Visualizations*, IEEE Symposium on Visual Languages, Los Alamitos, California (2000), pp. 67–68.

[2] Lucas, J., T. L. Naps and G. Rößling, *VisualGraph: a Graph Class Designed for both Undergraduate Students and Educators*, in: *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (2003), pp. 167–171.

[3] Naps, T., J. Eagan and L. Norton, *JHAVÉ: An Environment to Actively Engage Students in Web-based Algorithm Visualizations*, 31^{st} ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas (2000), pp. 109–113.

[4] Naps, T. L., G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger and J. A. Velázquez-Iturbide, *Exploring the role of visualization and engagement in computer science education*, ACM SIGCSE Bulletin **35** (2003), pp. 131–152.

[5] Rößling, G. and B. Freisleben, ANIMAL*: A System for Supporting Multiple Roles in Algorithm Animation*, Journal of Visual Languages and Computing **13** (2002), pp. 341–354.