Towards Tool-Independent Interaction Support

Guido Rößling, Gina Häussge
Department of Computer Science
Darmstadt University of Technology, Darmstadt, Germany

{guido, huge}@rbg.informatik.tu-darmstadt.de

Abstract

Interaction may be one key aspect for achieving improved learning outcomes using algorithm visualization software. Only a limited set of tools actually supports interaction, and the types of interaction supported are usually incompatible. In this paper, we present a tool-independent interaction support component that is easy to incorporate into existing systems.

1 Introduction

A group of eleven experts met in the course of ITiCSE 2002 to determine the role of visualization and engagement in CS education (Naps et al., 2003). They managed to isolate several key reasons why algorithm or program visualizations (abbreviated "AV" for the rest of this paper) have not been adopted as often as researchers in the field would hope. Chief among the reasons is *time* - the time needed to search for good examples, learning new tools, developing visualization and adapting materials to the use in the classroom.

In those cases where the use of AV software was evaluated, research has shown an important trend: learners who are actively engaged with the visualization technology have consistently outperformed learners who passively view visualizations (Hundhausen et al., 2002). One key part of the working group report is therefore the *engagement taxonomy*. The taxonomy defines six different levels of engagement in combination with AV materials:

1. *No viewing* – no use of AV technology,
2. *Viewing* – from passive to controlling the direction and pace of the animation, possibly including the use of different views or accompanying textual or aural explanation,
3. *Responding* – answering questions about the AV content presented by the system. This can for example include questions about the coding, efficiency, debugging or a prediction of the next step(s) in the algorithm,
4. *Changing* – modifying the AV content, for example by providing specific input data,
5. *Constructing* – building a new visualization of a given algorithm or data structure, usually based on educator-defined limitations,
6. and *Presenting* – presenting AV content to an audience for feedback and discussion. The AV content may or may not have been created by the learners themselves.

Figure 1 illustrates the dependency of the different levels of engagement, with *Viewing* acting as the base for the higher forms of engagement. *No Viewing* is outside the figure's window, as it does not entail any use of AV software.

The basic hypotheses of the working group can be summarized as follows:

1. There is no significant difference between *No Viewing* and *Viewing* - that is, learners do not benefit from purely passive use of AV tools.
2. Apart from the first two hierarchy steps *No Viewing* and *Viewing*, each higher step can result in statistically significant (and thus measurable) better learning outcomes than the previous steps.

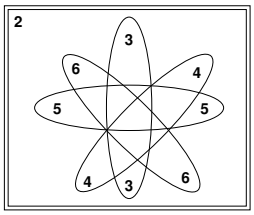

Figure 1: Possible Overlaps in the Engagement Taxonomy. The basic regions are 2 (*Viewing*), 3 (*Responding*), 4 (*Changing*), 5 (*Constructing*), and 6 (*Presenting*).

The working group report also includes a set of example experiments to verify these hypotheses. However, there is (at least) one obstacle for interested users: evaluating the different levels of engagement requires a tool which can handle the appropriate functionality. For example, to verify if *Responding* is more beneficial than *Viewing*, the AV tool used for the experiment must be able to handle question pop-ups during the visualization.

Currently, this type of interaction support is mostly restricted to highly context-specific tools such as *PILOT* (Bridgeman et al., 2000) or *Interactive Data Structure Visualization (IDSV)* (Jarc et al., 2000). Both systems are essentially restricted to graph problems and let the user select the next graph node or execution step for the given algorithm and graph.

The *MatrixPro* system (Karavirta et al., 2004) offers far-reaching interaction. Users can directly interact with given data structures and simulate underlying algorithm. The animation essentially illustrates the reaction of the underlying data structure to the user's graphical insert or delete operations. Users are not asked any questions during the animation, apart from the lead-in question outlining the task they have to accomplish. *MatrixPro* is difficult to place in the engagement taxonomy – it certainly addresses level 2 (viewing) and 4 (changing the values). It also supports parts of level 5 (constructing). The support for level 3 (responding) is still too limited to use *MatrixPro* for experiments that require this level of engagement.

2 Current Interaction Support in *JHAVÉ* Using Animal

JHAVÉ (Naps et al., 2000) is one of the few general-purpose AV systems that offer full interaction support. This includes free-text answers, multiple choice, true/false questions, and showing a documentation page in HTML format.

The interaction implementation for *JHAVÉ* contains an *infoFrame* class which is used to display questions and HTML documentation. The supported question types cover the standard elements *true / false*, *select m out of n*, and *fill in the blanks* (free-text). With a small effort, the AV system Animal could be adapted to use the same interaction front-end (Rößling and Naps, 2002).

The implementation works fine for the current AV systems incorporated in *JHAVÉ* - namely, *GAIGS*, *JSamba* and Animal (Rößling and Naps, 2002). However, the structure of the implementation prevents easy adoption into other systems. This is mostly due to the following three factors: notation, embedding, and portability.

First, the interaction components are defined in a scripting notation that has to be parsed. This effectively means that any AV system author interested in incorporating the interaction support has to modify the parsing component to address a set of other commands. How easily this can be accomplished depends on the structure of the AV system in question.

We would ultimately have n nearly identical parsing components in n different AV systems - where each author has to reinvent the wheel for his or her system. Furthermore, the notation used for the interaction components may not fit the notation used so far. For example, some systems use integer values as object IDs, whereas JHAVÉ's interaction support uses strings placed in double quotes.

Second, the interaction commands are directly embedded into the animation code. Both *JSamba* and ANIMAL use a scripting language to generate the visualization content. To ensure that the interaction elements occur at the right point in time, a specific place-holder is used. The animation ends with the actual definition of the interaction elements.

Finally, parts of the *infoFrame* code are system-specific and cannot easily be adapted to other systems. For example, *JHAVÉ* offers a *quiz for real* mode. In this mode, each new question freezes the animation and stores the answer (once it is given) over the WWW in the answer database on a central server. Other systems, such as *Interactive Data Structure Visualizations* (Jarc et al., 2000), use a different – and probably incompatible – approach for storing learner answers.

3 Designing a Tool-Independent Interaction Component

To address the problems listed in the previous section, we have to take three steps:

- separating (as much as possible) the definition of interaction elements from the animation itself,
- providing a general parser and graphical interface for interaction events,
- and offering a flexible evaluation back-end for handling answered questions.

Our new component requires only the invocation of the following two methods in the *avinteraction.InteractionModule* Java class:

interactionDefinition("location") defines the location of a file describing the interaction components. The structure of the file and indeed the type of location should be flexible. For example, *location* could be a scripting file on the local hard disk, an XML file on a Web server, or a dynamically generated resource anywhere on the Web.

Optionally, either the MIME type of the definition file or a concrete parser can be passed in as a second parameter. In this way, it is easy to ensure that nearly any implementation can work without touching the core implementation of the *avinteraction* package.

interaction("interactionID") invokes the interaction element with ID *interactionID* at this point of execution. The ID can be any arbitrary string, and thus may also encode contextual knowledge such as the exhibited "skill" of the learner (Rößling and Naps, 2002), provided that the input parser or interaction generator support this.

Both methods are easy to use and should therefore be easy to incorporate in most AV systems. Depending on the type of underlying AV system, the appropriate places in which the methods are invoked can be determined by embedding commands in the scripting notation, adding (non-visual) elements, adding appropriate declarations for declarative systems, or adding API invocations for API-based systems. This also holds for the determination of the command parameters. The only restriction is that the *interactionDefinition* command must appear before the *interaction* commands. Typically, the definition is placed in the animation header or among the first animation commands.

AV system developers may also need a flexible evaluation back-end that can store the learner's answers and possibly submit them to other evaluation engines. For example, answers for the *JHAVÉ* / ANIMAL cooperation have to be submitted to the web server for *JHAVÉ* in the *quiz for real* mode. For this end, we need a flexible interface that allows us to easily switch evaluation back-ends. This is relatively easy to do in Java if coded carefully.

Figure 2 shows the schematic structure of the proposed interaction component. The upper part shows how a single *interactionDefinition* and several *interaction* commands are sent to the interaction parser. After parsing the associated interaction definition file, the interaction

component has an internal representation of the visualization's interaction elements. The interaction parser is designed in modules to support customized definition file formats for defining the interaction components.

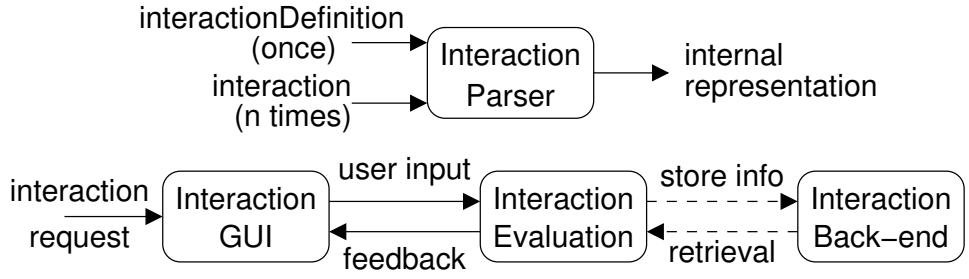

Figure 2: Schematic Overview of the proposed interaction component

The lower part of Figure 2 illustrates the typical data flow during interaction. The interaction request (with a concrete request ID) is resolved in the internal representation of interactions and then pops up the GUI, if not already visible. After a while, the user provides his or her input, which is sent to the built-in evaluation component. Feedback is then sent back to the user over the GUI.

Optionally, the learner's answer may also be sent over the interaction back-end to the appropriate "listener", typically a data base or learning management system. User data may also be incorporated into the evaluation of the answers, for example by dropping certain questions after the user has shown sufficient proficiency in the topic (Rößling and Naps, 2002).

4 Supported Features

The currently supported interaction types are identical to those offered by JHAVÉ / ANIMAL:

- *true / false* questions, containing one radio button for *true* and *false*, respectively,
- *free-text* questions, where the user can enter an arbitrary text as the answer. Note that this type of input is notoriously difficult to evaluate, due to the large number of possible answer formulations. For example, to indicate that the answer to a given question is "the first array element", the user could write "a[0]", "first element", "the first" or "7".

  For practical reasons, the evaluation of the free-text input is restricted to a case-insensitive string matching. Note that sub-string matching ("any expression that includes ...") may be more hindrance than help: clever learners will simply incorporate several possible answers into their text and hope that at least one of them matches. Even in graded tests, the educator may rely on the automatic assessment rather than checking the exact input for each submission. To alleviate this situation, the educator can define an arbitrary number of correct or acceptable answers.
- *multiple choice* questions, where the user selects a subset of the defined answers,
- *HTML documentation* links to further documentation or other pages.

Each question is set in a text area including scroll bars if necessary. The AV interaction designer can provide a comment for each possible answer to tell the users why their answer is correct or wrong. In the latter case, the answer may also give hints to the correct answer.

The package contains two default back-ends, both of which simply echo any user input on the standard output. The default parser is called *AnimalscriptParser* and parses interaction elements based on the notation used in JHAVÉ and ANIMAL (Rößling and Naps, 2002).

```
%Animal 2
interactionDefinition "http://www.algoanim.net/iav/interactionDemo"
array "a" (100, 100) length 5 int {3, 7, 8, 2, 12}
interaction "sortComplexity"
# further animation commands...
```

Listing 1: Example animation code including interaction components

Listing 1 shows a very brief example interaction script. In this listing, the command notation is nearly identical to the API calls, except for the missing parentheses and class names. Mapping the script input to API invocations is therefore trivial. The implementation also prevents "curious" learners from gleaning the correct answer from the animation code itself. The definition of the interaction elements is defined in a separate file which has to be parsed by the interaction manager – *not* by the AV system.

Figure 3 shows an example screen shot of the interaction element using the multiple choice answer defined in Listing 2. The user has selected two correct answers. The feedback for each selected answer is displayed at the bottom of the window. The system also comments if correct answers are missing or incorrect answers were chosen. For layout reasons, this feedback is not shown - but note the scroll bars of the text area at the bottom of the window.

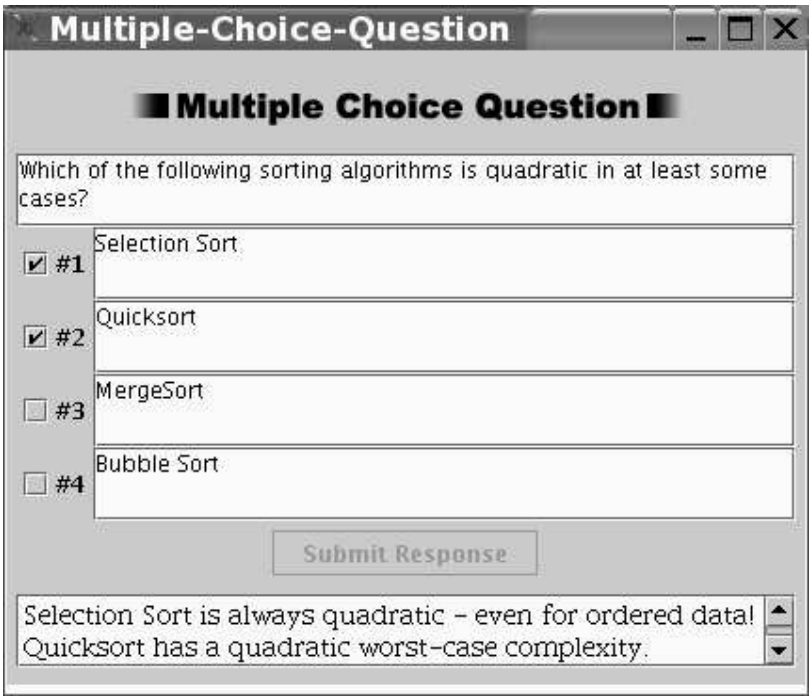

Figure 3: Example Interaction Screen Shot

Especially in long visualizations, users may become bored or even frustrated by repeatedly being forced to answer easy questions. On the other hand, the personal perception of the depth of understanding and the actual level of knowledge tend to differ. Additionally, not all questions are equally difficult. We should therefore enable good learners to skip questions that they have answered correctly sufficiently often.

Listing 2 shows the example code for the example definition file for the animation. Line 2 declares the actual points to be gained in this question. Line 3 declares the question as belonging to a group of questions named "general complexity", which may be repeated until the learner has correctly answered questions from this group two times (`nrRepeats 2`). After

this, further questions of this question group are skipped.

The actual question is defined in lines 5 to 6. The rest of the definition contains one block for each answer. Each block defines an answer text (lines 8, 14, 20, and 26), ended by *endchoice*. An answer can be given a specific number of points. Here, the understanding that *Quicksort* is quadratic in *at least some cases* gives two points, while the same answer for *Bubble Sort* and *Selection Sort* gives only one point. Each answer also has its own comment to help the learner's understanding (lines 12, 18, 24, and 30). The question ends with the definition of the set of correct answers in lines 33-35. Usually, there should be at least one correct answer and one incorrect answer.

```
mcQuestion "sortComplexity"
points 4
questiongroup "general complexity"
nrRepeats 2
"Which of the following sorting algorithms is quadratic in"
" at least some cases?"
endtext
"Quicksort"
endchoice
points 2
comment
"Quicksort has a quadratic worst-case complexity."
endcomment
"Bubble Sort"
endchoice
points 1
comment
"Bubble Sort is indeed quadratic."
endcomment
"Selection Sort"
endchoice
points 1
comment
"Selection Sort is always quadratic - even for ordered data!"
endcomment
"MergeSort"
endchoice
points 1
comment
"Mergesort has a worst-case complexity of O(n log(n))!"
endcomment
answer
1
2
3
endanswer
```

Listing 2: Example interaction script

There is also a default feedback for correct answers ("Yes, you are right!"), in case the author has not provided a more concrete answer comment. The number of possible free-text input values is effectively indefinite. The *comment* entry for this question type therefore acts as a "catch-all" for all possible incorrect answers. The system currently does not support the specification of expected "incorrect answers" with predefined reactions. In general, free-text

answers require some leniency to prevent learner frustration caused by giving a correct but unrecognized answer.

Multiple choice questions always randomize the order of the answers before they are displayed. This way, the same question asked four times may still require clicking in different places each time. The evaluation of the question ends with the assignment of points. The *points* command in line 2 defines the number of points given if the question has been correctly answered. For most question types, assigning points is a straightforward process. The interaction back-end, as shown in Figure 2, has to decide if and how the points for a given question or all questions are shown to the learner. The default back-end simply displays the number of points reached on the default Java output stream.

For multiple choice questions, the actual point number is determined by several parameters. A number of points can be defined for the whole question (see line 2 in Listing 2). Additionally, every answer option can have its own number of (positive) points (lines 10, 16, 22, and 30 in Listing 2). The number of points assigned to each selected correct answer is added to a separate sum of points. Similarly, the number of points assigned to a selected incorrect answer is deducted from this sum. In this way, partly incorrect answers can be translated into a decrease of already gained points. If no point value is given, no points are deducted from the point sum - this is especially useful for "radio button" (1:n) questions.

If the sum calculated in this way is less than zero, it is reset to zero. If a question is answered correctly by selecting exactly all correct answers, and the sum of points is still less than the total number of points, the total number of points possible is awarded. This can happen if the author does not ensure that the number of subpoints add up to the desired total points.

Developers can easily implement their own parser based on a simple interface. The new parser class should belong to the package *avinteraction.parser*. Making the parser visible for the system is achieved by modifying the ASCII-based configuration file *parser.config* that accompanies the package. Here, the MIME type of the input format has to be mapped to the parser class name in the notation shown in this example:

```
"text/animalscript" = "avinteraction.parser.AnimalscriptParser"
```

5 Summary and Further Work

We have presented the design for a tool-independent interaction component. Principally, this component can easily be incorporated into most existing AV systems based on Java or capable of invoking Java methods. For example, we finished a *proof of concept* embedding of our package into the *JAWAA* system (Akingbade et al., 2003) within just five hours. The largest part of this time was actually spent in determining the concrete classes to modify inside JAWAA and to figure out how the interactions could be integrated as "events" in JAWAA.

Due to the common GUI and evaluation component, AV system authors are not required to implement anything apart from the two simple interaction API invocations. More advanced or specific systems can also provide a special back-end to support answer feedback and advanced evaluation, for example for testing or scientific evaluation. They may also use special interaction parsers for other definition file formats, such as formats based on XML or interaction generation depending on database content. By using this modular structure in the component, we hope to ensure a high flexibility that enhances the component's usability.

Using the *avinteraction* component makes it far easier for AV system developers and users to design experiments using the *Responding* level of engagement. Such experiments could then be used to test the Working Group's hypothesis that responding to questions during a visualization may increase learning outcomes. Here, our package plays a key part in enabling not only AV system developers, but virtually *any* user to define interactions for a given system, and evaluate the way users interact and potentially benefit from this.

As further work, we plan to evaluate the ease of use of the AV interaction package. For this end, we need to convince authors of others tools such as *Jeliot* (Moreno et al., 2004) or *Matrix* (Korhonen, 2003) / *MatrixPro* (Karavirta et al., 2004). Any interested developer is encouraged to contact the first author of this paper in order to explore possible cooperations. Additionally, the system is freely available on `http://www.animal.ahrgr.de` under the link *Downloads*.

References

Ayonike Akingbade, Thomas Finley, Diana Jackson, Pretesh Patel, and Susan H. Rodger. JAWAA: Easy Web-Based Animation from CS 0 to Advanced CS Courses. In *Proceedings of the* 34^{th} *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2003), Reno, Nevada*, pages 162–166. ACM Press, New York, 2003.

Stina Bridgeman, Michael T. Goodrich, Stephen G. Kobourov, and Roberto Tamassia. PILOT: An Interactive Tool for Learning and Grading. *Proceedings of the* 31^{st} *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 139–143, March 2000.

Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing*, 13(3): 259–290, 2002.

Duane Jarc, Michael B. Feldman, and Rachelle S. Heller. Assessing the Benefits of Interactive Prediction Using Web-based Algorithm Animation Courseware. *Proceedings of the* 31^{st} *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 377–381, March 2000.

Ville Karavirta, Ari Korhonen, Lauri Malmi, and Kimmo Stålnacke. MatrixPro – A Tool for On-The-Fly Demonstration of Data Structures and Algorithms. In *Proceedings of the Third Program Visualization Workshop, University of Warwick, UK*, pages 26–33, July 2004.

Ari Korhonen. *Visual Algorithm Simulation*. PhD thesis, Department of Computer Science and Engineering, Helsinki University of Technology, 2003.

Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing Programs with Jeliot 3. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI 2004), Gallipoli (Lecce), Italy*, pages 373–380. ACM Press, New York, May 2004.

Thomas Naps, James Eagan, and Laura Norton. JHAVÉ: An Environment to Actively Engage Students in Web-based Algorithm Visualizations. *Proceedings of the* 31^{st} *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 109–113, March 2000.

Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. Exploring the Role of Visualization and Engagement in Computer Science Education. *ACM SIGCSE Bulletin*, 35(2):131–152, June 2003.

Guido Rößling and Thomas L. Naps. Towards Improved Individual Support in Algorithm Visualization. *Second International Program Visualization Workshop, Århus, Denmark*, pages 125–130, June 2002.