Journal of Visual Languages and Computing 21 (2010) 1–22

Contents lists available at ScienceDirect

Journal of Visual Languages and Computing

journal homepage: www.elsevier.com/locate/jvlc

A comprehensive taxonomy of algorithm animation languages

Ville Karavirta [a,*], Ari Korhonen [a], Lauri Malmi [a], Thomas Naps [b]

[a] *Department of Computer Science and Engineering, Helsinki University of Technology, Finland*
[b] *Department of Computer Science, University of Wisconsin, Oshkosh, USA*

ARTICLE INFO

Article history:
Received 27 February 2008
Received in revised form
14 May 2009
Accepted 23 September 2009

Keywords:
Algorithm animation
Algorithm animation language
Software visualization
Scripting language
Taxonomy

ABSTRACT

In this paper, we present a taxonomy of algorithm animation languages that augments Price's well-known taxonomy of software visualization. Whereas Price's taxonomy is directed to classifying features and characteristics of visualization systems, the new taxonomy focuses on evaluating current animation languages. The taxonomy can be used by algorithm visualization system designers as a tool to compare visualization system languages with each other as well as for designing and implementing new systems and language features. In addition, the taxonomy provides guidelines to the features that are needed for transferring animations between systems. This is an ongoing project that elaborates upon the work reported on in a briefer version of the taxonomy.

1. Introduction

Many algorithm animation tools have been developed over the past two decades. In recent years a key distinction has arisen between the notion of an *algorithm animation system* and an *algorithm animation language*. An algorithm animation system is a self-contained tool that may well use an internal representation that is not intended to be read by the end user nor to be interpreted by another system. MatrixPro [15] exemplifies such a system. Earlier systems of this sort included BALSA [3], TANGO [28], and many more. An algorithm animation language (hereafter referred to as an *AAL*) is the realization of a formal language description that specifies an algorithm animation. Such a language is intended to be written by hand or generated by a program and then animated by one or more algorithm animation systems. This approach is utilized by an increasing number of tools such as JHAVÈ [17], JAWAA [1] and Animal [21].

Price's well-known taxonomy of software visualization [20] focuses on software visualization and algorithm animation systems and hence is insufficient for the increasingly important role played by AALs. Our goal is to analyze animation languages. There is, however, considerable overlapping between the taxonomies and we will discuss this issue in some detail later.

Defining a general-purpose representation such as that required by an AAL was one of the goals of the ITiCSE working group in 2005. (*Developing XML-based tools to support user interaction with algorithm visualization* [19].) This is, however, not a trivial task due to the diversity of approaches in different visualization tools. The working group concluded that by providing XML definitions of the essential elements of a visualization, the aspects of program dependency and tight coupling of the animation objects with the visualization software can be eliminated. The criteria for a common algorithm animation language specification are that it must be able to express the data structures and graphical primitives used in the visualization, hierarchies of animation operations, and various style settings for the data structures and graphical primitives. The working group specified six different elements intrinsic to visualization systems: objects

* Corresponding author.
E-mail addresses: vkaravir@cs.hut.fi (V. Karavirta), archie@cs.hut.fi (A. Korhonen), lma@cs.hut.fi (L. Malmi), naps@uwosh.edu (T. Naps).

1045-926X/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jvlc.2009.09.001

(e.g., data structures) that are the focus of a visualization, graphical primitives (squares, circles, lines, etc.), transformations on graphical primitives (e.g., scaling, rotation), narration (text, graphics, audio), questions and feedback inserted in an animation, and meta-data that describe the content of an animation.

That working group's report appeared in conjunction with a much briefer taxonomy of AALs that was published in [14]. In this paper we have analyzed more languages and carried out that analysis at a much deeper level. Two developments have been most instrumental in leading to the need for this new analysis. First, there is the close relationship between the 2005 ITiCSE working group report [19] and the earlier taxonomy paper. Two authors of this expanded taxonomy paper were also authors of the working group report. That report was merely a starting point in an overall evolution of algorithm animation languages—indeed the report concluded by describing itself "as a framework for future research and development". Since the publication of the report, we have received considerable feedback from a variety of sources on the direction that research and development should take. That feedback is taken into account in the more comprehensive taxonomy that we describe here.

Second, the earlier version of the taxonomy presented it independently from its application in educational settings. In this new analysis we instead take into account the inescapable bond between algorithm animation and its deployment in education and learning environments. This linkage is recognized in Diehl's comprehensive examination of the entire field of software visualization [7], in which he states: "Most of the (algorithm animation) systems …have been developed mainly for educational purposes". While it is certainly possible to recognize other application areas for algorithm animations, such as facilitating communication between experts and optimizing and debugging programs, we feel it is proper and inevitable that the main focus of algorithm animation will continue to be that of an educational tool.

The rest of the paper is organized as follows. First, in Section 2 we argue for the increasingly important role that algorithm animation languages are likely to play—especially in educational settings. Then, in Section 3 we report on the analysis of a number of current animation languages. In Section 4 we define the taxonomy for categorizing and comparing the different AAL definitions. Finally, in Section 5 we draw some conclusions regarding future developments in the field.

2. The increasing importance of algorithm animation languages

Understanding the increasingly important role of algorithm animation languages requires that we examine more deeply the relationship between such languages and the algorithm animation systems that were the focus of Price's earlier taxonomy [20]. To do this, we use some definitions that draw on analogies from programming languages. An *algorithm animation language* is a *textual* language that is used to specify animations or visualizations. The language has a well-defined set of concepts, syntax and semantics, which are defined in the *language specification*. An *animation script* is a "program" written in some algorithm animation language. The animation script defines a set of visual objects and their relations and how they change dynamically.[1] A *visualization system* is a tool capable of interpreting an animation script and representing it in a visual form. The visualization tool may also allow many different ways of controlling and modifying the animation on the fly. In addition, there may be *translators* that parse some animation script and translate it to another visualization language.

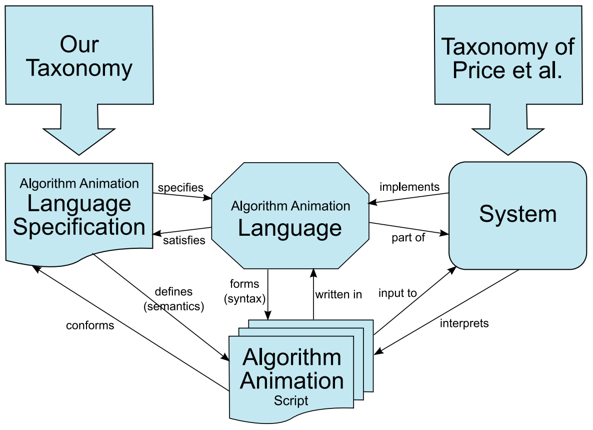

Fig. 1. Algorithm animation languages vs. algorithm animation systems.

Fig. 1 illustrates these concepts. Each AAL has a unique specification. This specification may be *implemented* by multiple visualization tools that are capable of interpreting the language script.[2] The tool may implement a language wholly or only partially. Moreover, a single tool may be able to interpret scripts written in several different animation languages.

Price's taxonomy concentrates on a conceptual analysis of the right hand side of the figure. Hence the language specifications are not a part of it. In contrast, our new taxonomy analyzes the left side of the figure, discussing the concepts and differences in various AALs and their implementations. Nonetheless, there is some overlap between the two taxonomies. For example, MatrixPro [15] has many AAL features available in its GUI, but the ASCII representation format used by MatrixPro supports only a few of them for export to other tools. Nor is the internal representation and the ability to save animations as serialized Java objects (a unique feature of MatrixPro) considered to be an animation language. In those situations where features in animation languages and animation systems do overlap, we have decided to adopt the terminology used in the Principled Taxonomy of Software Visualization by Price et al. [20] where applicable.

A useful analogy comes from the realm of programs that can be used to produce various media files and those

[1] We do not distinguish the cases where the script includes no dynamics but defines only a static visualization.

[2] Until recently there has been one-to-one correspondence as each language has been implemented as a property of some specific system.

programs that can then view such files. There are well-established formats for the contents of those files, and, because of that, there can be multiple programs that produce them and multiple programs that view them. This allows producers and viewers of such files to work within the context of one environment, using a consistent "look and feel", even though they are dealing with multiple file formats. We envision that something similar will occur in the world of algorithm animations. An algorithm animation script is essentially to be viewed as a multi-media presentation of an algorithm, usually for instructional purposes. Algorithm animation creators will want to work in the context of a single system to produce animation scripts even though such scripts may be written in a variety of scripting languages. Similarly students viewing the presentations of the algorithms will want to use a single viewing program with a consistent look-and-feel. Indeed, an industry-wide survey conducted by Bassil and Keller [2] indicated that the ability to import and export visualizations into various third-party programs was the biggest need in the field of software visualization. Particularly in educational applications, a survey cited in [18] reported that fewer than 10% of instructors regularly use algorithm animations even though over 90% of those instructors intuitively feel that algorithm animations aid students in understanding. Moreover, Hundhausen's meta-study of algorithm animation effectiveness [10] presents solid statistical evidence from a variety of experiments showing these instructors are correct in their intuition—algorithm animation does indeed work when it is employed properly. Then why do we have the apparent contradiction between instructors' awareness of a strategy that will help their students and their deciding not to use that strategy? One of the primary reasons cited in the survey results in [18] was the time it takes a student to learn how to use the algorithm animation tools. Over 90% of the respondents cited this as one of the primary reasons they did not actually use algorithm animations. If students only need to learn one algorithm animation viewing environment to view a broad variety of algorithm animations scripted in different languages, then this major constraint on the instructional use of algorithm animations will have been largely eliminated.

Our taxonomy can hence be seen as serving two valuable purposes. First, the details of our taxonomy are technical in nature and, as such, will be of most direct use to developers who are writing the software to produce and/or render algorithm animation scripts. For software that produces algorithm animation scripts, our taxonomy provides a unified view of the consequences of including or not including various aspects of the AAL specification. For developers writing renderers of algorithm animation scripts, the taxonomy informs them of semantics that may be omitted by the language they are trying to interpret and render. These are the direct benefits accruing to developers from the technical detail of the taxonomy. However, from the alternative perspective of the survey results cited above, it is clear that by helping these software developers, the groups who will benefit significantly from the increased availability of this software are the educators, students, and other end users who right now suffer from the lack of interoperability between animation systems.

Consider how this benefit is already starting to emerge in the educational community. At the time of Price's taxonomy and in the 5- to 10-year period following it, the state of the art had been that there was a one-to-one relationship among systems and languages. Now, however, this is starting to change rapidly. For example, the JHAVÉ algorithm visualization viewing environment [17] is capable of presenting algorithm animations in the GaigsXML [16], AnimalScript [25], Samba [27], and XAAL [12] AALs. Each such language is viewed as a JHAVÉ plug-in. By registering as a plug-in for JHAVÉ, a language's algorithm animation scripts are presented in an environment with a consistent look-and-feel for students. Moreover, by triggering events handled completely by JHAVÉ, the plug-in for the animation language automatically gains access to event-handlers capable of supplementing the graphics of the animation with learning support tools such as pop-up questions, synchronized pseudocode, hypertext document display, and audio accompaniment. JHAVÈ itself runs as a Web-startable Java application, hence giving students very convenient access to algorithm animations from four different algorithm animation languages—all with a consistent GUI and support structure that encourages increased student engagement. As the predictions of the 2006 ITiCSE working group [24] become increasingly realized, the role of AALs as plug-ins to hypertextbook learning environments and course management systems will lead to even more dramatic changes.

3. Animation languages

The results of the literary survey conducted for this paper are summarized in the following subsections. We discuss first the scope of the analysis: what kind of tools and languages will be analyzed. This is followed by a description of several different types of AALs and examples thereof.

3.1. Scope of the analysis

Algorithm animations can be generated and presented using a wide variety of tools and representations, ranging from codes processed by graphical processors to various drawing tools. The focus of this analysis, however, is to consider the algorithm animation languages that are capable of importing, exporting, and representing algorithm animations independently of any actual animation system. By independent, we mean a situation in which it could be possible to develop multiple systems that can use the same animation language even though the two systems do not share any code.

Even though algorithm animations can be created in many different ways, our focus in this paper is to analyze such languages that have specific support for concepts and operations that intrinsically belong to algorithm animations, such as data structure concepts or operations

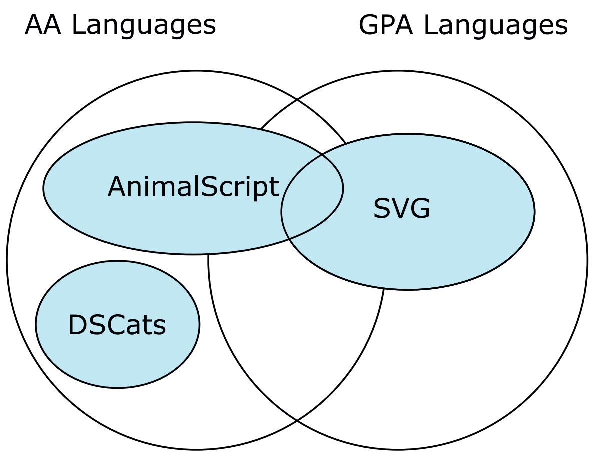

Fig. 2. Features of algorithm-animation languages versus those of general-purpose animation (GPA) languages and their overlap with the feature set of particular example languages.

like swapping items in an array. These languages often share many features with general purpose animation languages, such as SVG, but neither of these language groups is a true subset of the other group. Fig. 2 illustrates this. AALs may include features specifically designed for algorithm animation that are not available in general purpose animation languages (for example, the capability to automatically draw standard data structures such as arrays, trees, and graphs), whereas these general purpose languages have features that are not needed in algorithm animation. When considering specific systems we note that some systems fall into different parts of the same diagram. Thus, DSCats has no support for graphical primitives whereas AnimalScript has a wide support for them.

In the following, we explain why certain kinds of animation languages are included or excluded from the analysis.

The application is too general in its purpose. Drawing tools like *xfig* [26] can be used to generate static and dynamic animations. However, we are not interested in this broad area of tools for two reasons. These are general tools not designed for algorithm animations, and their internal representation of the graphical information is difficult or impossible to access.

The abstractions offered by the language are too low level. Graphical processors have specialized instruction codes for manipulations of graphical data. Even though animations will eventually be presented using such codes while they are displayed, these codes are not designed to be used directly from applications. Rather they are used through graphical toolkits, software libraries and device drivers. We omit all such codes, as well as the corresponding graphical software packages, because they work on concepts that are too low level, they lack support for algorithm animation operations, and the animation tools typically apply them through some other software layers.

Relative to Fig. 2, *the language's feature set lies almost entirely in the right domain*. Such languages can be used for presenting algorithm animations although they are not specifically designed for this purpose. Even though we do not consider them as AALs, we include one such language, SVG [31], in our comparison. This language acts as a reference point for the properties available in this class of languages/tools. We choose SVG as this reference point because it is comprehensive, relatively well-known, XML-based with a formal specification, and supported by an increasing number of modern browsers and mobile devices.

The abstractions of the language are too general in nature. Typical programming languages such as Java can be used for creating animations. We, however, consider them too general to be included in our comparison, and they also lack support for specific algorithm animation operations.

The language is too reliant on dependencies to particular code libraries. A number of algorithm animation tools are based on library calls that are inserted into the program code to be animated. Even though the system itself is very well fit for algorithm animations, the animation language itself does not satisfy our criteria of independence. These tools include, for example, Tango [28] and Polka [30] that use direct calls to graphical animation routines, and Leonardo [6] that includes animation commands into program comments in a special format. The animations are created through a dynamic process of compiling and executing the target program, and the language does not exist independently of the tool itself. Some tools, such as MatrixPro, allow storing animation information in a file using Java serialization. We consider that such a binary format is also not an independent language.

Program animation languages and visual programming environments. Finally, we also omit special languages that have been developed for teaching introductory programming, such as Alice [5], where complicated visual operations can be created in terms of a special language. The main goal of the tool is not in teaching algorithms but basic programming concepts. The same applies for visual programming languages.

We have attempted to include and discuss all languages that meet our criteria for an AAL. On the other hand, taxonomies such as this can never be complete due to the new functionality that is constantly added to the animation languages; thus for brevity, we do not cover all the languages in such a detail that could be possible. Taxonomies are intrinsically intended for extension in case more thorough evaluation is needed in certain situations, and we have taken this into account in our omissions as well. We also mention some evident ways to extend the current taxonomy in areas that are too extensive to be completely covered here. For example, some animation languages include a complete programming language that could have been analyzed by a taxonomy of its own.

3.2. Algorithm animation languages

AALs are tied to existing algorithm animation systems. They typically support both static (e.g., the state of the data structures) and dynamic visualizations (e.g., operations on the data structures). As examples, we will consider two different cases and include examples of the case we feel the language is better suited for. The example cases are the following.

- A case where a key is inserted into a binary search tree, and the search path during the insertion is highlighted.
- A case where an array is sorted using some sorting algorithm.

It should be noted that the script examples below do not comprise complete animations, but are only fragments of an entire animation script.

3.2.1. AnimalScript

AnimalScript [25] is a scripting language for the Animal [21] algorithm visualization system. AnimalScript provides support for graphical primitives such as arc, circle, ellipse, line, point, polygon, polyline, square, and text. Coordinates for the objects can be given either in absolute coordinates or relative to another object or location. AnimalScript supports also data structures array and list as well as code animation. Listing 1 shows how the binary search tree example case can be described using AnimalScript. Note, that AnimalScript is more suitable for demonstrating the array example. However, since Listing 2 shows the array example using programming constructs in AnimalScript2, we decided to show the BST example here.

AnimalScript2 [22] introduces programming constructs including loops, conditionals, arithmetic expressions, and integer variables. Listing 2 gives an example where the code sorts the array using bubblesort. It should be noted that examples with different data can be achieved by changing the data on the first line.

3.2.2. DSCats

DSCats [4] includes a command language for defining the animations. The language supports the definition of the structures, insert and delete operations, break points, annotations, and several options. An example of the BST insert is represented in Listing 3. As DSCats has automatic visualization, the highlighting of the search path cannot be specified in the language. Furthermore, DSCats uses numbers instead of alphabets.

3.2.3. GaigsXML

GaigsXML is an XML language for JHAVÉ-II [16]. Scripts consist of snapshots of the visualization at different times. The language supports data structures as well as has elements to describe documentation, pseudocode, and interactive questions. The animations can be viewed using the JHAVÉ algorithm visualization environment. Listing 4 gives an example of GaigsXML script by giving a snapshot of the final state of a BST example case. The complete animation of the example case would have preceding `snap` elements before the element shown in the example.

3.2.4. GraphXML

GraphXML [8] is a graph description language designed to be used between graph drawing and visualization

```
{
circle "c1" (44,16) radius 13 color blue filled fillColor white
text "t1" "M" (39,20) color (255, 0, 0) size 12
group "node1" "c1" "t1"
circle "c2" (26,53) radius 13 color blue filled fillColor white
text "t2" "A" (22,57) color (255, 0, 0) size 12
group "node2" "c2" "t2"
circle "c3" (62,53) radius 13 color blue filled fillColor white
text "t3" "S" (59,57) color (255, 0, 0) size 12
group "node3" "c3" "t3"
line "l1" (38,27)(31,41) color blue
line "l2" (49,27)(56,41) color blue
}
{
color "c1" type "fillColor" orange
color "c2" type "fillColor" orange
}
{
move "node1" to (39,3)
  ... moving the rest of the nodes and the lines
circle "c4" (53,90) radius 13 color blue filled fillColor white
text "t4" "F" (50,94) color (255, 0, 0) size 12
group "node4" "c4" "t4"
}
```

Listing 1. The BST example in AnimalScript using graphical primitives and basic animation.

```
array "values" (10, 10) length 5 int "3" "2" "4" "1" "8"
int i = 4;
arrayMarker "i" on "values" atIndex i label "i"
int j = 1;
arrayMarker "j" on "values" atIndex j label "j"
for (; i >=0; i--) {
  moveMarker "i" to position i within 5 ticks
  for (; j <= i; j++) {
     moveMarker "j" to position j within 5 ticks
     if (values[j-1] > a[j])
       arraySwap on "values" position i with j within 5 ticks
  }
}
```

Listing 2. A bubblesort example of programming constructs of AnimalScript2.

```
OPTION DETAILLEVEL 3
OPTION SPEED 4000
OPTION DS BINARY TREE
INSERT 20 30 10
INSERT 15
```

Listing 3. The BST example using DSCats command language.

```
<show>
  <snap>
    <title>Gaigs Example</title>
    <tree>
      <binary_node color="red">
        <label>M</label>
        <left_node color="red">
          <label>A</label>
          <right_node color="red">
            <label>F</label>
          </right_node>
        </left_node>
        <right_node>
          <label>S</label>
        </right_node>
      </binary_node>
    </tree>
  </snap>
</show>
```

Listing 4. GaigsXML example listing showing a snapshot of the final state of the BST example case.

software. It has basic elements to describe the graph. Interesting features of the language are the support for geometry, visual properties, and dynamic graphs. Listing 5 shows an example. Note that the comments in the listing represent parts that have been left out of the example. Although not shown in the example, it is possible to add another edit and highlight the search path.

3.2.5. *JAWAA*

JAWAA 2.0 [1] includes a scripting language for creating animations and viewing them over the web. JAWAA commands are divided into three types: primitive objects, action commands, and data structures.

Primitive objects in JAWAA are circle, rectangle, line, polygon, and oval. Action commands allow movements

```
<graph id="example">
  <style>
   <line tag="node" linestyle="solid" colour="blue"/>
   <fill tag="node" fillstyle="solid" colour="white"/>
   <line tag="edge" linestyle="solid" colour="blue"/>
   <fill tag="edge" fillstyle="node"/>
  </style>
  <node name="M">
    <position x="30" y="10"/>
    <size width="20" height="20"/>
  </node>
  <!--Similarly nodes A and S-->
  <edge source="M" target="A"/>
  <edge source="M" target="S"/>
</graph>
<edit action="replace" xlink:href="#example">
  <!--Specify the same graph as above with updated positions-->
  <node name="F">
    <position x="50" y="90"/>
    <size width="20" height="20"/>
  </node>
  <edge source="A" target="F"/>
</edit>
```

Listing 5. GraphXML example of the BST example case.

```
begin
circle c1 21 3 26 (0,0,255) (255,255,255)
text t1 29 20 "M" (255,0,0)
groupObject node1 2 c1 t1
circle c2 3 40 26 (0,0,255) (255,255,255)
text t2 12 57 "A" (255,0,0)
groupObject node2 2 c2 t2
circle c3 39 40 26 (0,0,255) (255,255,255)
text t3 49 57 "S" (255,0,0)
groupObject node3 2 c3 t3
line l1 28 27 21 41 (0,0,255)
line l2 39 27 46 41 (0,0,255)
end
begin
changeParam c1 bkgrd orange
changeParam c2 bkgrd orange
end
begin
moveRelative node1 8 0
  ... moving the rest of the nodes and the lines
circle c4 29 77 26 (0,0,255) (255,255,255)
text t4 40 94 "F" (255,0,0)
end
```

Listing 6. The BST example using JAWAA primitives and action commands.

and changes to objects or object groups. Listing 6 shows an example using primitive objects.

Data structures specified in JAWAA language are array, queue, stack, list, tree, and graph. These structures have the normal operations, for example, stack has the operations push and pop and queue has the operations dequeue and enqueue. However, according to documentation, trees do not work in the current version of JAWAA. Listing 7 shows the array sorting example in JAWAA.

3.2.6. *SALSA*

SALSA (Spatial Algorithmic Language for Storyboarding) [9] is a high-level language for creating low fidelity storyboards. SALSA enables layout and logic of visualization to be specified in terms of its spatiality among visual objects.

SALSA includes three data types: *cutout*, *position*, and *s-struct* (spatial structure). Cutout is a computer version of a paper cutout. Position represents a point in the coordinate space. A spatial structure is a region in which cutouts can be arranged according to a particular spatial layout pattern (for example, grid).

The 12 commands of SALSA can be divided into four categories: element creation and modification, conditional execution, iteration, and animation. Listing 8 gives an example of SALSA where a user inputted array is sorted using bubble sort.

3.2.7. *Samba*

Samba [29] is a scripting language that allows creation of named graphical objects and their subsequent modification by later commands in the script. Graphical

```
array a1 25 25 5.1 3 2 4 1 8 horz black white black
delay 900
begin
changeParam a1[0] bkgrd red
changeParam a1[1] bkgrd red
end
changeParam a1[0] swap a1[1]
begin
changeParam a1[0] bkgrd white
changeParam a1[1] bkgrd white
changeParam a1[2] bkgrd red
changeParam a1[3] bkgrd red
end
changeParam a1[2] swap a1[3]
begin
changeParam a1[2] bkgrd white
changeParam a1[3] bkgrd white
changeParam a1[4] bkgrd gray
end
  ... performing the rest of the swaps
```

Listing 7. The sorting example using JAWAA array data structure.

```
create array a1 with 5 cells
input elements of a1 as integers
set i to 0
while i < cells of a1
  set j to 4
  while j >= i + 1
    if a1[j-1] > a1[j]
      swap a1[j-1] with a1[j]
    endif
    add 1 to j
  endwhile
  add 1 to i
endwhile
```

Listing 8. A bubblesort example of SALSA. Note that the array is given as user input.

objects available are line, rectangle, circle, triangle, polygon, and text. The modification commands include, for example, commands to move and exchange objects, to change the colors, and to change text and visibility of objects. Many of the modifications can be done either as smooth animation or in one frame (discreet change). Listing 9 gives an example of the BST case as a Samba script.

Samba also has a support for multiple views by introducing two commands that can be used to define a view and setting the current view. In addition, Samba has many front-ends, for example, JSAMBA [27] and JHAVÉ [17].

3.2.8. XAAL

XAAL [13] is an XML language that is based on the report by the ITiCSE visualization working group [19] mentioned earlier. It uses the graphical primitive specifications of the working group. XAAL continues the work by specifying more of the topics related to algorithm animation, like data structures and their operations. XAAL allows the specification of animations on different

```
{
circle c1 0.34 0.84 0.03 blue1 outline
bigtext t1 0.325 0.83 0 red M
set node1 2 c1 t1
circle c2 0.16 0.47 0.03 blue1 outline
bigtext t2 0.155 0.46 0 red A
set node2 2 c2 t2
circle c3 0.52 0.47 0.03 blue1 outline
bigtext t3 0.525 0.46 0 red S
set node3 2 c3 t3
pointline l1 0.34 0.84 0.16 0.47 blue1 thin
pointline l2 0.34 0.84 0.52 0.47 blue1 thin
}
{
fill c1 half
fill c2 half
}
{
moverelative node1 0.05 0
  ... moving the rest of the nodes and the lines
circle c4 0.39 0.1 0.03 blue1 outline
bigtext t4 0.375 0.09 0 red F
}
```

Listing 9. The BST example using Samba command language.

```
<initial>
  <tree id="tree" root="node1">
    <structure-property name="name" value="Binary Search Tree"/>
    <structure-property name="class" value="matrix.structures.CDT.
        probe.BinSearchTree"/>
    <node id="node1"><key value="M"/></node>
    <node id="node2"><key value="A"/></node>
    <node id="node3"><key value="S"/></node>
    <edge from="node1" to="node2" directed="false"/>
    <edge from="node1" to="node3" directed="false"/>
  </tree>
</initial>
<animation>
  <insert target="tree"><key value="F"/></insert>
</animation>
```

Listing 10. The BST example case using XAAL data structures.

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <circle opacity="1.0" cx="34" cy="16" r="13" stroke="rgb(0, 0,
     255)" fill="rgb(255, 255, 255)">
    <animate attributeName="cx" fill="freeze" dur="2.0s" from="34.0"
         to="42.0" begin="4.0s"/>
    <animateColor attributeName="stroke" fill="freeze" dur="2.0s"
        from="rgb(0,0,255)" to="rgb(51,46,230)" begin="1.0s"/>
  </circle>
  <text opacity="1.0" x="29" y="20" fill="rgb(255, 0, 0)" font-
     family="Serif" font-size="12">M<!-- Animation --></text>
  <!-- Specification of two more nodes is similar. -->
  <line opacity="1.0" x1="28" y1="27" x2="21" y2="41" stroke="rgb(0,
      0, 255)" fill="none"/><!-- Animation -->
  <line opacity="1.0" x1="39" y1="27" x2="46" y2="41" stroke="rgb(0,
      0, 255)" fill="none"/><!-- Animation -->
  <text opacity="0.0" x="28" y="94" fill="rgb(255,0,0)" font-family=
     "Serif" font-size="12">C<animate attributeName="opacity" fill=
     "freeze" dur="2.0s" from="0.0" to="1.0" begin="4.0s"/></text>
  <!-- Adding new circle and line -->
</svg>
```

Listing 11. The BST example case using SVG shapes and animation.

levels of abstraction using graphical primitives or data structures. Listing 10 shows how the same BST example could be described using data structures.

3.3. Other languages

Algorithm animations are often described using graphical primitives, thus also graphical description languages are relevant here. Although we earlier limited our scope to AALs, we will introduce one general-purpose animation language, Scalable Vector Graphics (SVG) [31] for comparison.

Scalable Vector Graphics (SVG) is an XML language targeted for describing graphics. The key features that we are interested in here are the graphical shapes defined in SVG. These are rectangle, circle, ellipse, line, polyline, and polygon. Another interesting feature of SVG is the animation. Listing 11 gives the BST example case using shapes and animation. Note, that the comments in the listing represent parts that have been left out of the example for brevity.

4. Taxonomy

Using as background the survey of the AALs in the previous section, we now define a *taxonomy of algorithm animation languages* by which to evaluate them (Fig. 3).

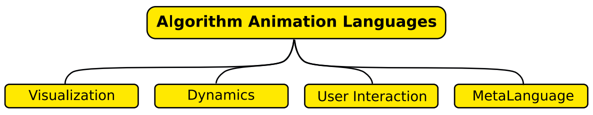

Fig. 3. Taxonomy of algorithm animation languages.

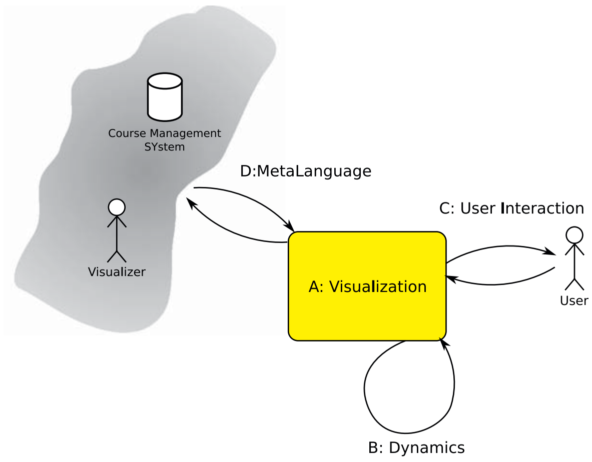

Fig. 4. Top-level categories of the taxonomy of algorithm animation languages.

At the top level of the taxonomy, the categories are A: *Visualization*, B: *Dynamics*, C: *User Interaction*, and D: *MetaLanguage*. The category *Visualization* considers the variety of supported object types, that is, the building blocks used in the animation as well as ways to position and style the objects. The category *Dynamics* describes the level and versatility of the language features that can be used to modify the Visualization and thus create the animation. The category *User Interaction* describes the level and type of interaction provided for the end user of animations. The category *MetaLanguage* inspects properties of the language not directly related to algorithm animation but still helpful in the process of creating useful animation content. These categories are illustrated in Fig. 4.

In the following sections, we will break down these categories of the taxonomy in more detail. In addition, we will evaluate the algorithm animation languages introduced in the previous section using this newly defined taxonomy. As in [14], we make frequent use of tables to summarize our findings. The reader should keep in mind that we evaluate the languages, not the systems: a feature might be available in a system, but not supported by the AAL.

4.1. A: Visualization

Category A: *Visualization* (see Fig. 5) measures the features of the language used to create static visualizations describing one state in the animation. It has three subcategories: A1. *Vocabulary*, A2. *Positioning*, and A3. *Style*.

4.1.1. A1. Vocabulary

Description: Category *Vocabulary* (represented in Fig. 6) describes the amount of supported object types. Basically, these are the building blocks used to compose the visualizations. Vocabulary has four subcategories: A1.1. *Data Structure Concepts*, A1.2. *Data Structure Components*, A1.3. *Graphical Primitives*, and A1.4. *Multimedia*.

Data Structure Concepts includes the different supported data structures available in the language. These include both linear structures (for example arrays, linked lists, stacks, and queues) and more complex non-linear structures (for example trees and graphs).

The *Data Structure Components* includes those components that data structures are composed of conceptually, as opposed to graphically. For example, trees and graphs are composed conceptually of nodes and edges whereas graphically a node could be rendered as a circle or rectangle. Similarly arrays are composed of array cells and indices while linked lists are composed of nodes and pointers.

The *Graphical Primitives* subcategory specifies the supported rudimentary geometric objects—rectangles, circles, lines, and so forth. Usually, there exist quite similar primitives with possibly different names. There are, however, languages where graphical primitives are not supported.

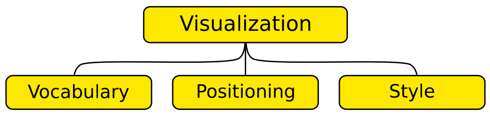

Fig. 5. Category Visualization.

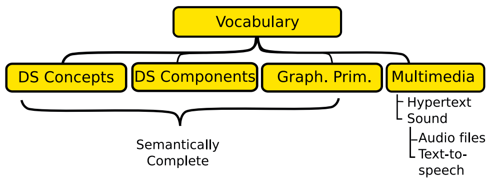

Fig. 6. Category Vocabulary.

A language that supports all of data structure concepts, data structure components, and graphical primitives can be said to have *semantic completeness*. This means that the language possesses enough information to link graphical primitives with the higher-level structures that the primitives may collectively be displaying. To see why this may be desirable, consider the following situation described in [19]. A student is viewing a visualization that displays an AVL-tree, and is asked a question that requires clicking on the tree node (circle) at which a rotation would occur when the data item 42 is added to the tree. For the visualization display system to determine whether the student has correctly answered this question, it would need to have considerable semantic information associated with the circles that are rendered on the display. Each of these circles would actually represent a tree node, so the complete visualization specification would need to define the context of each particular graphic circle in the AVL-tree. Without this contextual semantic information somehow embedded in the AAL script, the student's response could not be evaluated.

The *Multimedia* category describes whether or not the language supports the use of hypertext and sound resources in animations. Sound support may come in the form of static sound files or dynamic text-to-speech support. The latter, because it is generated at the same time as the graphic display, can be highly aware of nuances that are taking place in that display. However, the former will probably be of higher audio quality.

Evaluation: Table 1 indicates that only three AAL's—AnimalScript, JAWAA, and XAAL—could be considered semantically complete in the sense that we have defined earlier. We also observe that only two AAL's provide hypertext support for synchronized pseudocode. Sound support is rare, although a recent study in [11] indicates that it may have a significant beneficial impact on student learning.

4.1.2. A2. Positioning

Description: Category A2. *Positioning* (see Fig. 7) describes the different ways to position the objects in the animation. *Positioning* has five subcategories: A2.1. *DS Concept Layout*, A2.2. *Multiple Views*, A2.3. *Coordinates*, A2.4. *Dimensions*, and A2.5. *Grouping*.

Data Structure Concept Layout category has four subcategories: *Orientation*, *Visual size*, *Automatic layout*, and *Manual layout*. *Orientation* states how the language supports different orientations for data structures, for example, horizontal or vertical layout for arrays. *Visual size* measures the degree of support for changing the visual size of (parts of) data structures, for example, the size of a node or a whole graph. The degree to which a language supports visual size is particularly relevant for large data structures where effects such as selective zooming (for example, a "fish-eye" view) could be used to focus upon relevant parts of the structure. *Automatic layout* lists the structures for which the layout is automatic and *Manual layout* the data structures for which the layout must be manually specified. Although automatic layout can be considered a feature of the system, it is also relevant for the language. For example, how an end-user

Table 1
Evaluation of the languages in category Vocabulary.

Language	DS Concepts	DS Components	Graphical Primitives	Multimedia	
				Hypertext	Sound
AnimalScript, AnimalScript2	Array, list, graph	Array marker	Yes	Sync. pseudo.	No
DSCats	BST, AVL, B-tree	None	Text only	No	No
GaigsXML	Stack, queue, array, bargraph, linked list, tree, binary tree, graph	List item, tree node, edge, vertex	Text only	Web docs, sync. pseudo.	Audio files, text-to-speech
GraphXML	Graph	Node, data	None	No	No
JAWAA	Stack, queue, array-1D, array-2D, linked list, tree, graph	Nodemarker, list pointer, node	Yes	No	No
SALSA	Array-1D, array-2D	Array index, variable	Text only	No	No
Samba	None	None	Yes	No	No
XAAL	Tree, graph, array, list	Node, reference	Yes	Xhtml	Audio files
SVG	None	None	Yes	Yes	Audio files

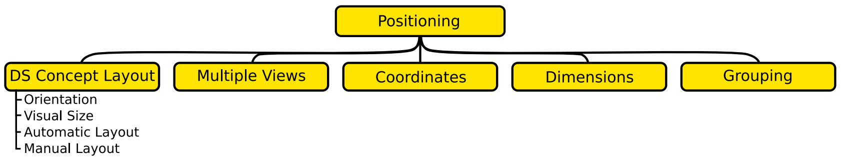

Fig. 7. Category Positioning.

such as a Computer Science instructor would use a language that supports automatic layout of a graph differs from a language that requires that same user to specify the layout providing the coordinates of the nodes in the graph.

Multiple Views measures the support for multiple views in the language. This can be, for example, support for several different canvases that can be painted on, or multiple synchronized views of the same data structure.

Coordinates list the various ways the coordinate information can be specified, and the available coordinate systems. For example, an object can be positioned using absolute coordinates or relative to some other location or object.

Dimensions state the maximum number of dimensions usable when positioning the objects. Usually, this value is either two or three, although we use 2.5 to reflect a layering capability in the language.

Grouping indicates whether or not the AAL allows the objects to be grouped together in strategic fashion.

Evaluation: The languages are evaluated in category *Positioning* in Table 2. DSCats and SALSA have only automatic layout and positioning, thus the coordinates criterion is not applicable. *Grouping* capability seems to be related to A1.2. Graphical Primitives. The only exception is GraphXML that has no graphical primitives, but still supports grouping (of elements in a graph).

AnimalScripts and XAAL support depth to control how overlapping objects are drawn on the screen. Samba supports a similar feature by allowing objects to be moved backward and forward from the viewer. Thus, the dimension is 2.5 in the evaluation of these languages.

4.1.3. A3. Style

Description: The category A3. *Style* (see Fig. 8) measures the variety of styling options available in the language. By styling, we mean setting the style of the objects in the language's vocabulary. Style is divided into seven subcategories: A3.1. *Colors*, A3.2. *Fill style*, A3.3. *Font*, A3.4. *Line style*, A3.5. *Opacity*, A3.6. *Shape*, and A3.7. *Stylesheets*.

Colors is used to describe the supported color space of the language. Usually, the languages support some predefined colors and colors given as RGB-values.

Fill style states the different fill style options. These can be, for example, gradient, pattern, and solid.

Font is used to examine the variety of means to specify the typography of the text objects used. Measures of this category are *family*, *size*, and *variants*. Family describes the different font families usable in the language, if any. This can be, for example, Serif or monospaced. Size states if the size of the font can be changed and variants lists the different font variants available in the language. Variants can be, for example, bold or small caps.

Line style states the support for different line style options. These can be, for example, dashed, pattern, solid, and width.

Opacity states whether or not the language supports the opacity of objects. If opacity is supported it is possible to make some objects partially transparent.

Shape considers whether or not the language supports specifying the shape of some objects. This can be, for example, specifying the shape of nodes in a tree to be rectangles instead of circles.

Table 2
Evaluation of the languages in category Positioning.

Language	**DS Concept Layout**				**Multiple Views**	**Coordinates**	**Dimensions**	**Grouping**
	Orientation	Visual size	Automatic layout	Manual layout				
AnimalScript, AnimalScript2	Yes	Yes	Arrays	Lists, graphs	Unsync. multiple canvases	Absolute, relative	2.5	Yes
DSCats	No	No	BST, AVL, B-tree	No	No	N/A	N/A	No
GaigsXML	No	Yes	Yes	Optional for graph	No	Absolute	2	No
GraphXML	No	Yes	Yes	Graph	No	Absolute, coordinate transformations	3	Yes
JAWAA	Yes	Yes	Stack, queue, array	Linked list, tree, graph	No	Absolute	2	Yes
SALSA	No	No	Array	No	No	N/A	N/A	No
Samba	N/A	N/A	N/A	N/A	Unsync. multiple canvases	Absolute	2.5	Yes
XAAL	Yes	No	Yes	Optional for some	No	Absolute, relative	2.5	Yes
SVG	N/A	N/A	N/A	N/A	No	Absolute, relative	2	Yes

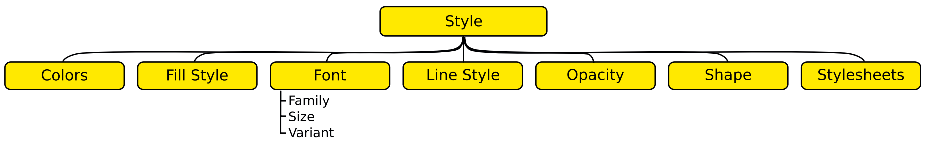

Fig. 8. Category Style.

Table 3
Evaluation of the languages in category Style.

Language	**Colors**	**Fill style**	**Font**			**Line style**	**Opacity**	**Shape**	**Stylesheets**
			Family	Size	Variant				
AnimalScript, AnimalScript2	Predefined	Solid, none	Yes	Yes	Bold, italic	Color, arrowheads	No	No	No
DSCats	Automatic	N/A	No	No	No	No	No	No	No
GaigsXML	Predefined and RGB	Solid	No	Yes	No	No	No	No	No
GraphXML	Predefined and RGB	Solid, none, background, image	No	No	No	None, solid, dashed, dash-dotted, dotted, width	No	No	Yes
JAWAA	Predefined and RGB	Solid, transparent	No	Yes	No	Color	No	Node	No
SALSA	No	N/A	No	No	No	N/A	No	No	No
Samba	Predefined	Solid, half, none	Yes	Yes	No	Width, color	No	No	No
XAAL	Predefined and RGB	Solid, none	Yes	Yes	Bold, italic	Width, color, dashed, arrowheads	Yes	Yes	Yes
SVG	Predefined, RGB	Solid, gradient, pattern	Yes	Yes	Weight, small-caps, italic, oblique, stretch	Yes	Yes	Yes	Yes

Stylesheets examines the support for stylesheets in the language. A Stylesheet is a style definition (e.g. colors and fonts) defined once and applied on multiple objects. The criterion for stylesheet support is that the stylesheets have to be re-usable among objects.

Evaluation: From Table 3 it can be seen that the languages are biased towards two groups: those that support styling and those that do not. DSCats and SALSA do not offer styling functionalities due to the automatic layout and styling, while the rest of the languages do.

The number of predefined colors and fonts available in Samba depends on the installed X windowing system. The fonts used in SVG document are the most flexible, and can be defined by the creator in the SVG document.

Stylesheets and opacity (see Table 3) are rare features in algorithm animation languages. Note that GraphXML does support the specification of styles for all nodes in a graph. However, these are not full-fledged stylesheets since they cannot be re-used or extended. Again, SVG has the most flexible styling options.

4.2. B: Dynamics

The category B: *Dynamics* (see Fig. 9) describes the features of the language that can be used to modify the visualization and thus create the animation. *Dynamics* has five subcategories: B1. *Data Structure Concept Operations*, B2. *Sequencing*, B3. *Timing*, B4. *Animation Effects*, and B5. *Programming Constructs*.

4.2.1. B1. Data Structure Concept Operations

Many algorithm animations are intended to visualize high-level operations on abstract data types like stacks, queues, dictionaries, trees, and graphs. Other animations may be designed to provide a much deeper granularity and focus on illustrating how such high-level operations are implemented. This category describes the functionality to modify ADTs with high-level operations and also to specify how to display the implementation of such high-level operations (if that is desired). There are three subcategories: B1.1. *ADT Operations*, B1.2. *Data Structure Implementation Operations*, and B1.3. *Data Structure Component Operations* (see Fig. 10).

The subcategory *Abstract Data Type Operations* focuses on the ADTs furnished by the language and the "high level" operations available for those ADTs. For example, a stack has push and pop high-level operations, a list has insert and remove high-level operations, and a graph has add-edge and remove-edge high-level operations.

The subcategory *Data Structure Implementation Operations* focuses on the more intricate details of the fashion in which the high-level operations may be implemented. For example, push/pop for a stack and insert/remove for a list may be implemented with array or linked list techniques, and add-edge/remove-edge for graphs may be implemented with adjacency lists or an adjacency matrix.

We realize that the distinction we make between ADT Operations and Data Structure Implementation Operations is a distinction that exists along a continuum rather than one that fits neatly into two mutually exclusive categories. The separation of high-level ADT Operations from underlying implementation techniques is an issue that has caused endless discussion and debate in the development of object-oriented libraries for data structures. Our intention here is not to be prescriptive about where a given AAL *must* be categorized in this regard but rather to give a perspective on where we think that language lies on the continuum that spans these two subcategories. For example, a language like AnimalScript is rich enough in its vocabulary that it covers both ends of that continuum. On the other hand a language like DSCats works much more at the ADT Operation end. The information provided in an animation script in a language like DSCats offers very little capability for the viewer of the animation to "drop down" a level in the degree of viewing detail and see how a particular operation of that ADT is actually implemented. For example, if the viewer were watching an insert operation on a dictionary, there would be no option to go deeper and see that operation carried out as insertion into a hash table with a particular collision-processing strategy or to see the insertion carried out on an underlying red-black tree. A language focusing on the ADT Operation end of the continuum simply does not have the semantic richness to illustrate this. This does not mean that someone writing a renderer for that language could not choose to provide such a deeper view of a possible implementation technique, but there would be significantly more work to do because the language is not providing any hints about how this is to be done.

The important thing from the perspective of our algorithm animation taxonomy is to recognize that there is a difference in the granularity of the two types of operations. For some animations, it may be desirable to shield the viewer from seeing implementation-oriented operations, and, in such cases, it is not important whether the algorithm animation language offers such operations. In other situations, we may want the viewer to focus on learning these implementation details or to at least have the option of dropping down to a more detailed view. In these situations, if the data structure implementation operations are not specified in the animation language, it will be impossible for the renderer to provide such views without somehow creating the context for those operations completely on its own without the benefit of context given by the animation script.

The *Data Structure Component* subcategory of Data Structure Operations is intended to indicate support for manipulating the components of data structures, for example, array indices and nodes in trees and graphs.

Evaluation: We can see from Table 4 that those languages working largely from graphic primitives

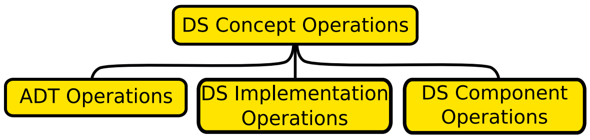

Fig. 10. Category Data Structure Concept Operations.

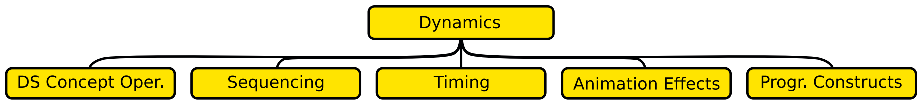

Fig. 9. Category Dynamics.

Table 4
Evaluation of the languages in category Data Structure Operations.

Language	ADT Operations	DS Implementation Operations	DS Component Operations
AnimalScript, AnimalScript2	ArrayPut, ArraySwap	ClearLink, SetLink	MoveArrayMarker
DSCats	Insert, Delete	None	Deletetype
GaigsXML	None	None	None
GraphXML	None	Remove, replace	None
JAWAA	Push, pop, enqueue, dequeue	ConnectNodes, swap array items	MoveMarker, target a listpointer, array index on/off
SALSA	None	Swap	Populate, move
Samba	None	None	None
XAAL	Insert, delete, search	Create, remove, replace	None
SVG	None	None	None

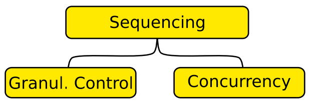

Fig. 11. Category Sequencing.

(Samba and SVG) cannot offer *Data Structure Operations* because they lack the semantic completeness that is necessary to carry out the animation of such operations. Even those languages that do offer a variety of *Data Structure Implementation Operations* tend to focus on operations for linear structures such as arrays and lists.

4.2.2. B2. Sequencing

Sequencing inspects the support for structuring the animation operations to be animated sequentially and/or simultaneously. *Sequencing* has two subcategories: B2.1. *Granularity Control* and B2.2. *Concurrency* (Fig. 11). *Granularity control* indicates whether or not the sequential animation operations can be selectively and strategically skipped for a viewer whose understanding of the algorithm may be sufficient to make seeing too much detail undesirable. For example, a viewer already knowledgeable about linked list operations could choose to omit seeing the details of these operations in an animation of an algorithm that employed a linked list as one of its data structures. *Concurrency* determines whether the language supports definition of concurrent operations in animations.

Evaluation: The languages are evaluated in category *Sequencing* in Table 5. As can be seen, many of the systems support granularity control, but some of them only through concurrent operations (that is, implicitly). The only language that supports both features explicitly is XAAL.

4.2.3. B3. Timing

Timing examines the support for timing of the animation. Timing includes, for example, the possibility to set the duration and start time of an animation operation, as well as animation speed. Although Timing is certainly related to Sequencing (B2), it also has impact on Animation Effects (B4).

Table 5
Evaluation of the languages in categories B2 Sequencing and B3 Timing.

Language	Sequencing		Timing
	Granul. control	Concurrency	
AnimalScript, AnimalScript2	No	Yes	Delay, duration
DSCats	Yes	No	Play speed
GaigsXML	No	No	No
GraphXML	Yes	No	No
JAWAA	No	Yes	Delay
SALSA	No	No	No
Samba	No	Yes	Delay
XAAL	Yes	Yes	Delay, duration
SVG	No	Yes	Delay, duration, min, max, repeat, key times

Evaluation: As indicated in Table 5, *Timing* possibilities are available in some AALs. Typically, supported features are setting the delay between operations or duration for a particular operation. However, SVG is even more versatile as the animation operations can be set a minimum or maximum duration, number of repeats, repeat duration, key times, etc.

4.2.4. B4. Animation Effects

Category B4. *Animation Effects* measures the various ways the graphical features of an object can be modified. It has two subcategories: B4.1. *Attributes*, and B4.2. *Transformations* (see Fig. 12).

Attributes measures the *Style* and *Visibility attributes* like in the two subcategories. *Style attributes* describes the dynamic support for animation effects such as changes in color, fill color, and line style. Similar to this, in some AALs the visibility of objects can be modified during the animation.

Transformations measures the ways the geometry of an object can be modified. The subcategories *rotate*, *scale*, and *translate* express the typical geometric transformations. *Rotate* and *scale* indicate whether the objects can be dynamically changed in terms of rotations and scaling, respectively. *Translate* examines how the position of an object can be changed.

Evaluation: As can be seen from the evaluation in Table 6, all the AALs can be roughly divided into two subcategories: those that support many animation effects and those that do not support it at all. Typical dynamic style attributes include color, background color, fill style, and some sizing attributes. Typically a mechanism to alter visibility is supported if at least some styling attributes are supported as well. *Remove* in case of JAWAA is in parenthesis to indicate that the operation cannot be undone, thus this attribute is supported only partially. In SVG and XAAL, visibility can be controlled through opacity, which makes this attribute more versatile than just on/off.

Transformations can be supported for any object and group of objects or only partially (indicated by parenthesis). For example, in JAWAA, rotations apply only for direction of list pointers and in AnimalScript for polyline/polygon and subtypes. The scaling of (only) single objects is supported in JAWAA, but not in Samba, in which, however, the entire screen can be zoomed. Finally, movement of objects is typically supported in terms of absolute or relative coordinates.

4.2.5. B5. Programming Constructs

Programming Constructs measure the degree of support for features similar to programming languages. By this we mean parts of the animation specification that can be used in calculations but that are not necessarily visualized. For example, in AnimalScript2, the *Programming Constructs* can be used to implement different sorting algorithms (see Listing 2 in Section 3.2). The script (language) does not contain a single trace of the algorithm to be visualized, but more powerful programming constructs, such as loops, can be used to describe the trace in general. We divide this category into two subcategories: B5.1. *Elements* and B5.2. *Control Flow* (see Fig. 13).

Elements are used to define the syntactical parts of an animation. For example, the typical constructs in many programming languages include *declarations*, *expressions*, *assignments*, and *types*. *Declarations* can be, for example, variable, function, or class declarations. *Expressions* can be arithmetic expressions and comparisons. *Assignments* can be evaluated expressions that are attached to variables. Finally, *Types* can be, for example, boolean, integer, or user defined. The idea is not to make an exhaustive analysis of a programming language, but to give a hint on what kind of programming constructs are currently available in the animation languages. This category can be extended in the future by adopting an additional taxonomy focusing only on programming languages.

Control flow measures the execution semantics of an animation language, that is, what kind of control structures the language supports to determine in which order the individual syntactical elements are executed. Again, these control structures are very similar to those in programming languages: sequentially executed statements, branching, loops, and (recursive) subroutines. *Branching* refers to structures that can result in conditional execution (that is, choice) of certain sequences. *Loops* are structures that provide a mechanism to repeat (that is, iteration over) a sequence. Finally, *Subroutines* refer to the ability to divide the animation code into smaller self-contained units that can be invoked from other units. If the unit can call itself, it supports recursive subroutines as well.

Evaluation: As can be seen from Table 7, programming constructs are not that general in algorithm animation languages. Typically scripting languages have their ways to refer to the entities on the screen by providing a unique ID for each entity. However, only few of the most recent languages like AnimalScript2 and SALSA have a support

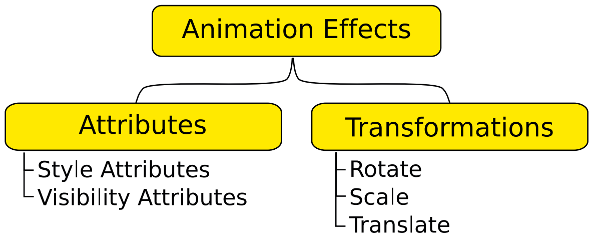

Fig. 12. Category Animation Effects.

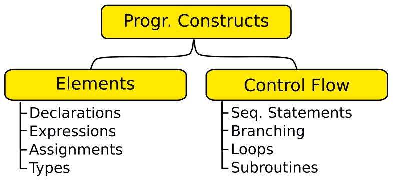

Fig. 13. Category Programming Constructs.

Table 6
Evaluation of the languages in category Animation Effects.

Language	**Attributes**		**Transformations**		
	Style	Visibility	Rotate	Scale	Translate
AnimalScript, AnimalScript2	Yes	Show/hide	(Yes)	No	Yes
DSCats	None	None	No	No	No
GaigsXML	None	None	No	No	No
GraphXML	None	None	No	No	No
JAWAA	Yes	(Remove)	(Yes)	(Yes)	Yes
SALSA	None	No	No	No	No
Samba	Yes	Toggle	No	(Yes)	Yes
XAAL	Yes	Show/hide, opacity	Yes	Yes	Yes
SVG	Yes	Opacity	Yes	Yes	Yes

Table 7
Evaluation of the languages in category Programming Constructs.

Language	**Elements**				**Control Flow**			
	Declarations	Expressions	Assignments	Types	Seq. statements	Branching	Loops	Subroutines
AnimalScript & AnimalScript2	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Dscats	None	No	No	No	Yes	No	No	No
GaigsXML	Element names	No	No	No	No	No	No	No
GraphXML	IDs/names	No	No	No	No	No	No	No
JAWAA	Element IDs	No	No	No	Yes	No	No	No
SALSA	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Samba	Visual primitive IDs	No	No	No	Yes	No	No	No
XAAL	Element IDs	No	No	No	Yes	No	No	No
SVG	Yes	Yes	Yes	Yes (boolean, strings, custom types)	Yes	Yes	Yes	Yes

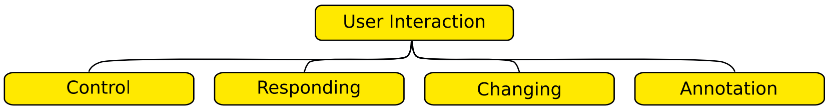

Fig. 14. Category Interaction.

for manipulating them through expressions and control flow structures such as branching and loops. None of the actual AALs, however, have a proper support for modularity through subroutines, yet it is possible to be implemented such as in SVG.

4.3. C: User Interaction

Description: The category *User Interaction* describes how the language supports interaction between the user and the animation. We must emphasize the distinction here between the interaction provided by visualization tools and interaction supported by the animation languages. This distinction is not always clear, as the language implementations may be tied to visualization systems. However, the key issue in this regard is that the language specification includes interactive features and actions that can be stored into a file containing the animation script, regardless of how a system presents them to the user.

User Interaction has four subcategories, as depicted in Fig. 14. The categories correspond rather closely to the different levels of the *engagement taxonomy* of algorithm visualization [18], i.e., levels *viewing, responding, changing and presenting*.

1. *Control* denotes the supported interaction to control the execution of the animation. Such features include, for instance, pausing the animation, and browsing it stepwise or continuously. A particularly important aspect when the animation is used in a learning environment is whether the language's description of the animation offers enough information to allow viewers to actually step backwards through the animation when they become confused. For example, if the language's description has broken down the animation into "frames" or a comparable notion, then the renderer for the language should be able to allow backward stepping. However, if insufficient semantic information is provided in the language specification, then stepping backwards is likely to be impossible.
2. *Responding* denotes language features for defining interactive questions for the users. These may include textual input, multiple choice questions or selecting items graphically. The animation, however, cannot be changed through such dialog. Rather the dialog is intended to assess the viewer's understanding of the algorithm as it is presented in the animation. Note that, if the language is not semantically complete as defined in our discussion of vocabulary, then the ability to select items graphically in any meaningful way will most likely be impossible. Hence choices made in defining the vocabulary of the language will greatly influence the ability of the language to engage in dialog with users.
3. Subcategory *Changing* includes language features that support modifying the algorithm visualization. For example, the input for an algorithm can be asked from the user in terms of entering data values, selecting or positioning nodes or drag-and-dropping items on the screen.
4. Subcategory *Annotation* represents a kind of explanatory comment or drawing that could be added to the script by the student who is viewing an animation. Such a comment is analogous to highlighting or adding annotations in the side margins of a book as it is being read. They are put there to help the learner better

Table 8
Evaluation of the languages in category Interaction.

Language	Control	Responding	Changing	Annotation
AnimalScript, AnimalScript2	No	Dialog	No	No
DSCats	Pause	No	No	No
GaigsXML	No	Dialog	No	No
GraphXML	No	No	No	No
JAWAA	No	No	No	No
SALSA	No	No	Input	No
Samba	No	No	No	No
XAAL	No	No	No	No
SVG	Yes	Yes	Yes	Yes

understand the animation the next time it is viewed. One can envision this being particularly helpful when the learner is going back to a previously viewed animation to review for an exam or when the animation is being viewed asynchronously by several students in a collaborative learning endeavor.

The above categories *Control*, *Responding*, *Changing*, and *Annotation* are not inclusive, that is, a language may support one or more of them in any order.

Evaluation: The languages are evaluated in category Interaction in Table 8.

In general, interaction features in AA languages are not common. Several languages have no support to interaction at all: Samba, JAWAA, GraphXML and XAAL. Thus, all interaction is left to the tool implementing the language.

DSCats is the only language supporting *Control*. AnimalScripts are extended to support pop-up questions during the animation [23], thus the languages support engagement level *Responding*. In GaigsXML such questions are a core feature. SALSA includes commands to get input data from the user, thus it is supporting level *Changing*. However, although it is conceivable that SALSA input dialogs could be used for *Control* or *Responding*, they are not intended for these purposes, and current SALSA animations do not use them in this fashion. None of the current AA languages support *Annotation*.

This result makes a strong contrast with SVG. Since SVG animations can include arbitrary ECMAScript code, it can be considered to support any kind of interaction.

4.4. D: MetaLanguage

Description: The category *MetaLanguage* (see Fig. 15) describes the support of features and properties that are not directly related to algorithm animation but are still useful in the animation creation process. The subcategories are D1. *Comments*, D2. *Debug*, D3. *Extensible*, D4. *Localization*, D5. *Metadata*, D6. *Import/Export*, and D7. *Syntax Specification*.

Subcategory *Comments* states whether or not a language supports comments. This is a comment authored by the designer of the visualization—as opposed to the annotation mode of interaction, which is authored by the viewer of the visualization and was described in Section 4.3. Such a comment would be analogous to a comment in a programming language. That is, the comment would not appear in normal viewing of the animation but would instead be there for clarification in trying to understand the text of the script from a developer's or designer's perspective. However, in certain circumstances the designer of the visualization may want to see their comments along with an animation—analogous to the fashion in which a lecturer may desire to see their "speaker's notes" along with the slides in a presentation file.

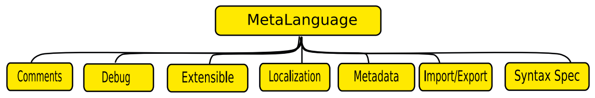

Fig. 15. Category MetaLanguage.

The *Debug* subcategory lists the support for debugging animations. This can be, for example, printing text to the standard output.

The subcategory *Extensible* indicates whether or not the language can be extended with reasonable effort. Such language is, for example, XML that can be extended by definition. For a language to be extensible, it must have been designed for that purpose by the language designers. We can draw an analogy from software systems to clarify this distinction. A software system is not considered to be extensible merely because it is implemented in Java and the source code is available. Yet, if there are well-defined and documented interfaces that can be implemented to add new commands to the language, the language is considered extensible. The situation is similar for AALs—the key question is whether or not the language was designed to facilitate extendibility.

Localization indicates whether or not the language supports localization of the animations. Usually, this is done by allowing textual data to be included in multiple languages. However, different instructors may typically use slightly different terminology. Thus, another example could be support (e.g., macros for terms) for this kind of "dialects". In general, any such support for integrating the material for a particular environment would be included here.

Metadata describes the additional information that can be included in animations. However, it is not enough for the language to support comments that can be used to include arbitrary information. The language must have some commands or tags to describe the information as well. This metadata can be, for example, information about the author, subject, or difficulty of the animation.

The *Import/Export* facilities of the language referred to here are not related to the previous user interaction category. Rather here we are concerned with the language's ability to conveniently communicate with external resources such as course management tools, hypertextbooks, and automated assessment systems—see [24]. For example, if the language supports the dialog subcategory under the interaction category, then the implication is that the user has answered questions about the animation during the course of viewing it.

An automated assessment system would need to know the responses that the student gave to the questions, so that the student's understanding of the material could be analyzed and perhaps reported to their instructor. The Import/Export facilities of the language would dictate how the algorithm animation interfaced with the automated assessment system to transmit these student responses.

The *Syntax Specification* of a language refers to the formal definition of the syntactical structure of the language. Although, in theory, we suppose that all AALs could have a formal language specification in something analogous to BNF notation, in practice such a specification may not actually exist. When it does not exist, the task of writing a parser for such a language is clearly more difficult. Hence relative to Syntax Specification, AALs can be broken down into those which are accompanied merely by a textual description of the language syntax and those for which a more formal specification is provided. Those which have such a formal specification can be further broken down into those whose formal specification is XML-based (XML DTD or Schema) and those providing a formal specification by BNF or something similar.

Evaluation: Table 9 shows the evaluation of the algorithm animation languages in category MetaLanguage. It is apparent from the evaluation that overall AAL support in this category is relatively sparse. From the AALs evaluated, only a small number offer support for Debug, Localization, and Metadata. Moreover, even in those situations where support is offered, it is very limited. Import/Export functionality is not offered in any AAL. It is only offered in the general-purpose animation language SVG. ANIMALSCRIPT's localization is an undocumented feature, thus this is indicated in the parentheses.

5. Conclusion

Since the early 1980s, many algorithm visualization tools have been developed to support teaching and learning core computer science topics such as data structures and algorithms. Despite the long history and the large number of systems available, disappointingly no breakthrough has taken place that can be attributed to the widespread use of such in education. In that sense, these tools have failed the educational community. Several reasons for this were revealed in the survey carried out by the ITiCSE working group in 2002 [18]. Among the top five impediments identified in the survey, three are related to this work: "Time it takes to learn the new tools" (90% of respondents listed this), "Time it takes to develop visualizations" (90%), and "Time it takes to adapt visualizations to teaching approach and/or course content" (79%). All these challenges can be tackled by promoting the importing, exporting, and sharing of content among algorithm animation systems.

From teacher's point of view, an ideal solution would be learning only one single visualization system that can be used to customize and adopt third party animations as well. In practice, this has been practically impossible due to lack of techniques to transfer animations from one visualization system to another. The ITiCSE visualization working group in 2005 [19] attacked this problem by creating a draft XML specification for algorithm animations. Such an intermediate language would support both (1) adapting visualizations made by others to one's specific needs, (2) sharing animations with other people, and (3) building animation repositories. One example language partially implementing the specification is XAAL [12,13].

In this paper, we continue to support this work by defining a taxonomy of AALs (depicted in its entirety in Fig. 16), and applying it to evaluate a set of current languages. As a result, we have a more detailed overview of the features and properties available in them. Our evaluation could be summarized by stating that animation languages can be broadly divided into two categories: languages emphasizing graphical primitives and languages emphasizing data structures. However, such a broad division belies the complexity of the issues involved, and the taxonomy we have presented takes this complexity into account.

The presented taxonomy can be used in several different ways. First, it summarizes in a concise way the multitude of features available in current animation languages, and clarifies the relations among these features. Second, the taxonomy can be used as a tool for comparing different languages with each other, and understanding their strengths and limitations. Third, it provides guidelines for features that must be taken into account in transferring animation information from one system to another.

Finally, the taxonomy can be used as a reference for future development of AALs. It cannot only indicate

Table 9
Evaluation of the languages in category D: MetaLanguage.

Language	Comm.	Debug	Ext.	Local	Meta.	I/E	Spec.
ANIMALSCRIPT, ANIMALSCRIPT2	Yes	Yes	Yes	(Yes)	Yes	No	Informal BNF
DSCats	Yes	No	Yes	No	No	No	Textual
GaigsXML	Yes	No	Yes	No	No	No	XML DTD
GraphXML	Yes	No	Yes	No	Yes	No	XML DTD
JAWAA	No	No	No	No	No	No	Textual
SALSA	Yes	No	No	No	No	No	Textual
Samba	Yes	Yes	No	No	No	No	Textual
XAAL	Yes	No	No	Yes	Yes	No	XML Schema
SVG	Yes	No	Yes	Yes	Yes	Yes	XML DTD

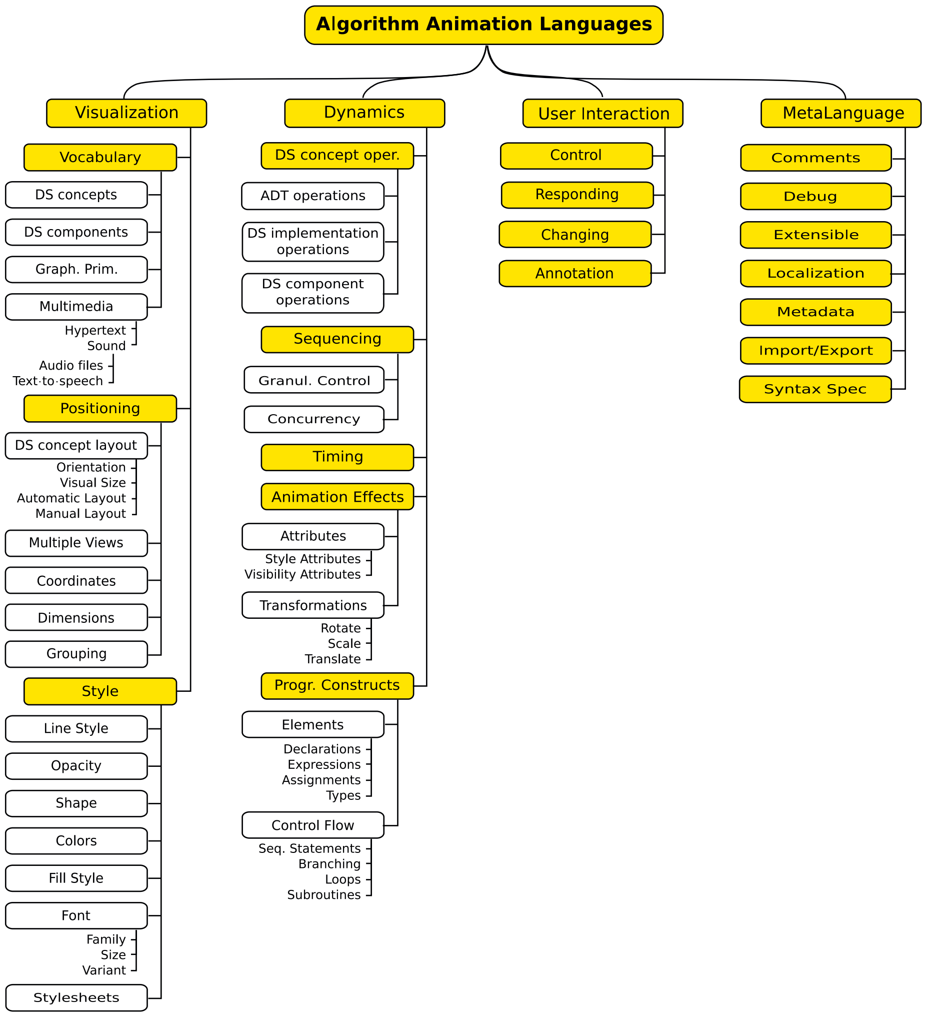

Fig. 16. Complete taxonomy.

what kinds of features are desirable when designing AALs, but also features that presently are very much under-utilized in existing AALs. We summarize some such features here.

- Table 1 summarizing AAL *Vocabulary* indicates that there is much still to be done in integrating algorithm animations with multimedia support. It has long been established that text accompanying animation and video is better presented in spoken fashion than in written [32]. A recent experiment [11] confirming the value of supplementing algorithm animations with audio accompaniment serves to confirm this prior evidence. In light of the ITiCSE 2006 Working Group report on merging interactive visualizations with hypertextbooks and multimedia [24], work in this area offers much promise in Computer Science education.
- Table 2 also indicates that multiple views are rarely being employed in current AALs.
- Table 3 on *Style* indicates that the potential offered by stylesheets to provide animation viewers with a consistent look-and-feel is rarely taken advantage of.
- Table 4 in the section on *Dynamics* indicates that current AALs offer virtually no support for ADT Operations on non-linear data structures such as trees and graphs. Clearly providing such support in the future will make it much easier to develop animations of the more sophisticated algorithms that typically employ non-linear structures.
- Tables 4 and 5 in the section on *Dynamics* indicate that present AALs are very limited in their ability to offer viewers explicit control of their ability to see multiple granularities of detail and concurrent operations during an animation. With viewers hence “locked in” to the same level of detail, they can become bored by having to see too many details or lost by not seeing enough.
- Table 7 on *Programming Constructs* in AALs indicate that no present AALs can match the richness of the programming constructs offered in SVG. Recalling

that SVG has been included in our taxonomy not as an AAL, but only to illustrate the contrasts between AALs and general purpose animation languages, it is clear that AALs might be able to profit greatly from borrowing aspects of what SVG does in this regard.

- Once AALs extend their capabilities in regard to *Programming Constructs*, they will change from mere scripts to real programming languages. That will enable them to not only be used to trace an instance of an algorithm but rather to actually express algorithms that consequently produce animations.
- Table 8 on *Interaction* indicates that, despite the overwhelming evidence (see [10,18]) that increasing student engagement via user interaction improves learning effectiveness, few AALs offer much support for interaction.
- Table 9 on *MetaLanguage* indicates that, despite widespread international interest in using algorithm animations, few AALs are equipped to deal with localization issues.
- Given the trend toward integrating algorithm animations into hypertextbooks and course management systems envisioned in [24], only SVG, which is not a true AAL, offers support for importing and exporting data to these other systems.

These represent a small subset of the information regarding future developments to be gleaned from the taxonomy. We are sure that readers will no doubt find much more, often very appropriate to their research and teaching interests.

Previous research has provided taxonomies (e.g. [20]) for evaluating visualization systems. Our work augments these efforts by presenting a taxonomic evaluation of algorithm animation languages. There are many reasons why a new taxonomy is needed, but of central importance is that such a taxonomy allows us to study languages and systems separately. Algorithm animation languages can provide functionality that goes beyond the basic GUI-based functionality covered in system taxonomies. For example, the algorithm animation language can include an extensive programming language that can be used to program the animations, or the language can have properties that are not shown in the GUI such as comments and metadata. However, we also recognize that some systems do not have any kind of algorithm animation language even though they might provide very rich GUI functionality or the system properties are covered only partially. Thus, the taxonomy can reveal the lack of functionality in the animation language. In addition, animations created or modified in such systems cannot be transferred to other systems until there is an agreement how to support all the relevant information available in the animations. Thus, the focus of our taxonomy is on the animation and content, not on the system properties.

We recognize that the taxonomy is not complete in the sense that it could cover all the features in all the available animation languages. Thus, as with all taxonomies, there is a constant need for evolution in the future as important new features become available. For this reason we have made an up-to-date version of the taxonomy available online[3] hoping that others would also contribute to it.

Acknowledgements

We gratefully acknowledge insights and commentary offered by Guido Rößling of the Technical University of Darmstadt during our work on this paper. They have immensely improved the final product.

References

[1] A. Akingbade, T. Finley, D. Jackson, P. Patel, S.H. Rodger, JAWAA: easy web-based animation from CS0 to advanced CS courses, in: Proceedings of the 34th SIGCSE technical symposium on Computer science education, SIGCSE'03, ACM Press, 2003, pp. 162–166.

[2] S. Bassil, R. Keller. Software visualization tools: survey and analysis, in: Proceedings of the 9th International Workshop on Program Comprehension, 2001, IWPC 2001, 2001, pp. 7–17.

[3] M.H. Brown, R. Sedgewick, A system for algorithm animation, in: Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'84, ACM Press, 1984, pp. 177–186.

[4] J. Cappos, P. Homer, DsCats: animating data structures for CS2 and CS3 courses, Technical Paper published online, 2002. Available online at: 〈http://www.cs.arizona.edu/dscats/dscatstechnical.pdf〉.

[5] S. Cooper, W. Dann, R. Pausch, Using animated 3D graphics to prepare novices for CS1, Computer Science Education 13 (1) (2003) 3–30.

[6] P. Crescenzi, C. Demetrescu, I. Finocchi, R. Petreschi, Reversible execution and visualization of programs with LEONARDO, Journal of Visual Languages and Computing 11 (2) (2000) 125–150.

[7] S. Diehl, Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software, Springer, New York, 2007.

[8] I. Herman, M.S. Marshall. GraphXML—an XML-based graph description format, in: Graph Drawing, 2000, pp. 52–62.

[9] C.D. Hundhausen, S.A. Douglas, Low-fidelity algorithm visualization, Journal of Visual Languages and Computing 13 (5) (2002) 449–470.

[10] C.D. Hundhausen, S.A. Douglas, J.T. Stasko, A meta-study of algorithm visualization effectiveness, Journal of Visual Languages and Computing 13 (3) (2002) 259–290.

[11] W. Hürst, T. Lauer, E. Nold, A study of algorithm animations on mobile devices, in: SIGCSE '07: Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education, New York, NY, USA, ACM, 2007, pp. 160–164.

[12] V. Karavirta. XAAL—extensible algorithm animation language, Master's Thesis, Department of Computer Science and Engineering, Helsinki University of Technology, December 2005. Available online at: 〈http://www.cs.hut.fi/Research/SVG/publications/karavirta-masters.pdf〉.

[13] V. Karavirta, Integrating algorithm animation systems, in: Proceedings of the 4th Program Visualization Workshop (PVW 2006), Electronic Notes in Theoretical Computer Science, vol. 178, Amsterdam, The Netherlands, Elsevier Science Publishers B.V., 2007, pp. 79–87.

[14] V. Karavirta, A. Korhonen, L. Malmi, Taxonomy of algorithm animation languages, in: SoftVis '06: Proceedings of the 2006 ACM Symposium on Software Visualization, New York, NY, USA, ACM Press, September 2006, pp. 77–85.

[15] V. Karavirta, A. Korhonen, L. Malmi, K. Stålnacke, MatrixPro—a tool for on-the-fly demonstration of data structures and algorithms, in: Proceedings of the 3rd Program Visualization Workshop, The University of Warwick, UK, July 2004, pp. 26–33.

[16] T. Naps, M. McNally, S. Grissom, Realizing XML-driven algorithm visualization, in: Proceedings of the 4th Program Visualization Workshop (PVW 2006), Electronic Notes in Theoretical Computer Science, vol. 178, Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 2007, pp. 129–135.

[3] URL: 〈http://svg.cs.hut.fi/aaltaxonomy/〉

[17] T.L. Naps, JHAVÉ: supporting algorithm visualization, Computer Graphics and Applications, IEEE 25 (5) (2005) 49–55.

[18] T.L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, J. Ángel Velázquez-Iturbide, Exploring the role of visualization and engagement in computer science education, SIGCSE Bulletin 35 (2) (2003) 131–152.

[19] T.L. Naps, G. Rößling, P. Brusilovsky, J. English, D. Jarc, V. Karavirta, C. Leska, M. McNally, A. Moreno, R.J. Ross, J. Urquiza-Fuentes, Development of XML-based tools to support user interaction with algorithm visualization, SIGCSE Bulletin 37 (4) (2005) 123–138.

[20] B.A. Price, R.M. Baecker, I.S. Small, A principled taxonomy of software visualization, Journal of Visual Languages and Computing 4 (3) (1993) 211–266.

[21] G. Rößling, B. Freisleben, ANIMAL: a system for supporting multiple roles in algorithm animation, Journal of Visual Languages and Computing 13 (3) (2002) 341–354.

[22] G. Rößling, F. Gliesche, T. Jajeh, T. Widjaja, Enhanced expressiveness in scripting using AnimalScript 2, in: Proceedings of the 3rd Program Visualization Workshop, The University of Warwick, UK, July 2004, pp. 10–17.

[23] G. Rößling, G. Häussage, Towards tool-independent interaction support, in: Proceedings of the 3rd Program Visualization Workshop, The University of Warwick, UK, July 2004, pp. 110–117.

[24] G. Rößling, T. Naps, M.S. Hall, V. Karavirta, A. Kerren, C. Leska, A. Moreno, R. Oechsle, S.H. Rodger, J. Urquiza-Fuentes, J.A. Velázquez-Iturbide, Merging interactive visualizations with hypertextbooks and course management, SIGCSE Bulletin 38 (4) (2006) 166–181.

[25] G. Rößling, M. Schüler, B. Freisleben, The ANIMAL algorithm animation tool, in: Proceedings of the 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'00, Helsinki, Finland, ACM Press, New York, 2000, pp. 37–40.

[26] B.V. Smith, xfig—a drawing tool for the X Window System. Available online at: ⟨http://epb.lbl.gov/xfig⟩.

[27] J.T. Stasko, Jsamba–java version of the SAMBA animation program. Available online at: ⟨http://www.cc.gatech.edu/gvu/softviz/algoanim/jsamba/⟩.

[28] J.T. Stasko, TANGO: a framework and system for algorithm animation, IEEE Computer 23 (9) (1990) 27–39.

[29] J.T. Stasko, Using student-built algorithm animations as learning aids, in: The Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education, San Jose, CA, USA, ACM Press, New York, 1997, pp. 25–29.

[30] J.T. Stasko, E. Kraemer, A methodology for building application-specific visualizations of parallel programs, Journal of Parallel and Distributed Computing 18 (2) (1993) 258–264.

[31] W3C, Scalable vector graphics (SVG) 1.0 specification ⟨http://www.w3.org/TR/SVG⟩, Sept. 2001.

[32] C. Wetzel, P. Radtke, H. Stern, Instructional effectiveness of video media, Lawrence Erlbaum Associates, Hillsdale, NJ, 1994.