The GAIGS-JHAVÉ Visualization System

The GAIGS-JHAVÉ Development Team

December 7, 2007

Contents

Chapter 1

What is GAIGS?

The GAIGS (Generalized Algorithm Illustration via Graphical Software) is an algorithm visualization scripting language that captures and renders snapshots of the state of an algorithm at interesting events–critical points in its execution.

Chapter 2

Core GAIGS XML Scripting

2.1 Overall Script Organization

The series of snapshots to be rendered by GAIGS is represented in XML. While GAIGS uses the file extension .sho for storing and reading its visualizations, the files are simply .xml files under a different name. When GAIGS loads one of these .sho files, the XML contained within is validated against its DOCTYPE, if one is provided. The built-in structures are all checked against the file gaigs_sho.dtd, and these are the structures that will be covered first.

(This introduction to GAIGS will assume the reader has a basic understanding of XML and DTD's.)

(Additional Note: It is now possible to use the "GAIGS Support Classes" to annotate the interesting events in your script-producing programs and completely avoid having to directly deal with the XML that is produced. See 3 for more information on these classes.)

2.2 Documentation and Pseudocode

2.3 Structures

The "data type definition" for GAIGS script, as specified in gaigs_sho.dtd

The root element of a .sho file is the "show".

```
<!ELEMENT show (snap+, questions?)>
```

A show consists of one or more snaps, optionally followed by questions. Questions will be covered later, but they are an important way of ensuring that someone viewing an algorithm visualization will be actively participating.

Here is the definition of a "snap":

```
<!ELEMENT snap (title, doc_url?, pseudocode_url?,
                (tree|array|graph|stack|queue|linkedlist|bargraph)*,
                question_ref?)>
```

title: The title element is simply #PCDATA. It can consist of multiple lines of text, and these lines of text appear centered at the top of a snapshot.

doc_url: The URL of the text that can be viewed in the Info tab as the visualization is running.

pseudocode_url: This is the same as the doc_url element, except the text can be viewed under the pseudocode tab when running a visualization. (If the Webserver you are using supports PHP, extensive support is provided to dynamically highlight lines of code – see Section 5.1)

structures: After the title and the two optional URLs comes zero or more structures. (These are all implemented in Java and descend from the abstract class StructureType – see Section 4.1.) Each of these structure types is defined in the DTD. They will be discussed next.

question_ref : Finally comes the question_ref. The question_ref element is empty, and has one CDATA attribute: "ref". This corresponds to a question element's "id" (covered later).

Here is an example of the most basic visualization GAIGS can produce:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE show PUBLIC "-//JHAVE//DTD GAIGS SHO//EN" "gaigs_sho.dtd">

<show>
  <snap>
    <title>Hello World</title>
  </snap>
</show>
```

This will produce a show with a single snapshot, containing nothing more than the title, "Hello World":

2.3.1 A structure's "name" element

All built-in structures have an optional "name" element. It contains nothing more than #PCTEXT. When drawing multiple structures to different areas of the screen, the structure's "name" element provides a simple and basic way of labeling the structure. However, if the structure is drawn in or near default bounds (the unit square), the structure's name may collide with the title for the entire snapshot as they are both drawn in the same way.

2.3.2 A structure's "bounds" element

All the built-in structures have an optional bounds element that can be used to position and resize structures. Here is the definition of the bounds element:

```
<!ELEMENT bounds (EMPTY)>
<!ATTLIST bounds x1 CDATA #REQUIRED
                 y1 CDATA #REQUIRED
                 x2 CDATA #REQUIRED
                 y2 CDATA #REQUIRED
                 fontsize CDATA "0.03">
```

The first four attributes define the coordinates of a rectangle that the structure will consider the bounds of where it is allowed to draw itself. The x1,y1 pair corresponds to the lower-left corner, and the x2,y2 pair corresponds to the upper-right corner. (It should be noted that not all features are rendered distortion-free when the height-to-width ratio is not one-to-one. For example, having a scaled height of 1.0 but a scaled width of 0.5 may create distortion in a vertical direction.) After the four coordinate attributes comes an optional fontsize attribute. Scaling a structure into a quarter of the area of the screen may make text so small as to be unreadable, but increasing the fontsize will solve that problem. The fontsize defaults to 0.03, which is in a rough sense 3% of the height of the GAIGS window at the default zoom level.

2.3.3 Colors in GAIGS

The built-in structures allow nodes, cells, and connecting lines to be colored independently from each other. The colors can be selected from a predefined set of colors, or through hexadecimal notation. The predefined colors are: white, black, red, green, blue, yellow, magenta, light blue

The hex format is a '#' character followed by six hex digits. The hex digits describe the color in standard RGB fashion: #RRGGBB.

```
#000000  <- black
#FF0000  <- bright red
#00AA00  <- green
#000055  <- dark blue
#888888  <- grey
#FFFFFF  <- white
```

2.4 The Built-in Structure Types

2.4.1 The stack, queue, and linkedlist structure types

The stack, queue, and linkedlist structure types all have the same syntax:

```
<!ELEMENT stack      (name?, bounds?, list_item*)>
<!ELEMENT queue      (name?, bounds?, list_item*)>
<!ELEMENT linkedlist (name?, bounds?, list_item*)>
```

The name and bounds are common to all structures. The stack, queue, and linkedlist structures all have zero or more list_items, each of which represents an entry in the data structure. A list_item is simply:

```
<!ELEMENT list_item (label)>
<!ATTLIST list_item color CDATA "#FFFFFF">
```

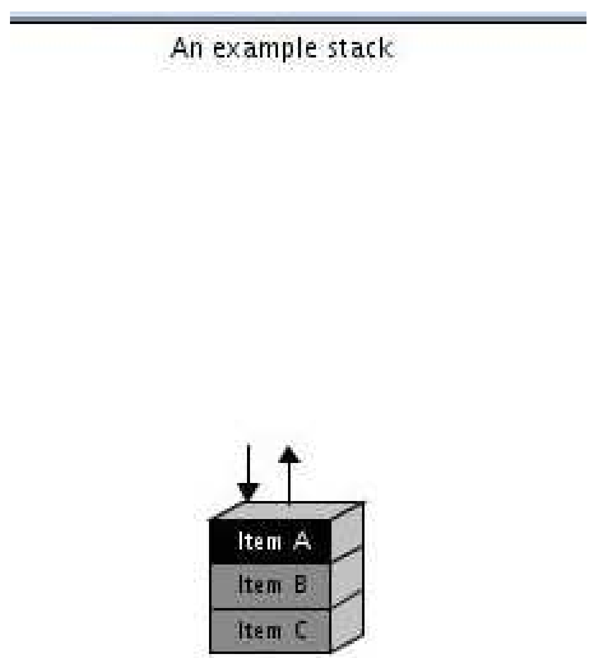
This describes a "label" for the cell (element containing only #PCDATA), and a CDATA attribute "color" describing the background color of the cell. The color of the label text is automatically determined to make the text readable.

The list_item given first will serve as the top of a stack, the head of a linkedlist, or the front of a queue. The subsequent list_items then come in order with the last one serving as the bottom, tail, or back of the data structure.

Here is an example stack. Changing the structure's tag (and the closing tag) name from "stack" to "queue" or "linkedlist" will be sufficient to produce a different data structure.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE show PUBLIC "-//JHAVE//DTD GAIGS SHO//EN" "gaigs_sho.dtd">

<show>
  <snap>
    <title>An example stack</title>
    <stack>
      <list_item color="#000000">
        <label>Item A</label>
      </list_item>
      <list_item color="#888888">
        <label>Item B</label>
      </list_item>
      <list_item color="#888888">
        <label>Item C</label>
      </list_item>
    </stack>
  </snap>
</show>
```

Given this code, GAIGS produces a show consisting of a single snapshot, titled “An example stack”. A three-element stack is centered on the drawing area with Item A on top, B in the middle, and Item C on the bottom. This stack has colored items, with Item A’s background color as black and the other two’s color as a shade of grey.



2.4.2 The array structure type

The array structure type is defined as follows:

```
<!ELEMENT array (name?, bounds?, row_label*, column_label*, column*)>

<!ELEMENT row_label (#PCDATA)> <!-- put in empty titles if you want to skip titling some rows --
<!ELEMENT column_label (#PCDATA)>

<!ELEMENT column (list_item*)>
```

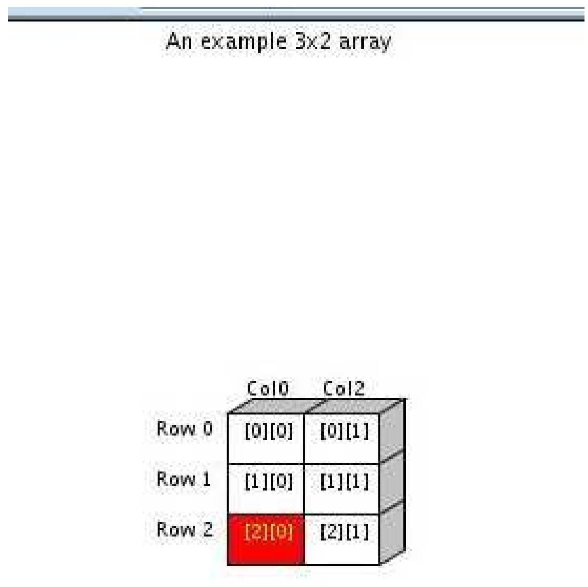
As with all other built-in structures, the first two elements are the optional name and bounds. The next set of entries is zero or more row_labels (which are element containing only #PCDATA). These are drawn to the left of the array, with the first one appearing at the top, or row index 0. The column_labels are handled similarly, with the first appearing at the left, or column index 0. The final entry in an array is a set of zero or more columns. Each column consists of zero or more list_items, which are the same as the list_items used in stacks, queues, and linkedlists.

Here is an example 3x2 array.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE show PUBLIC "-//JHAVE//DTD GAIGS SHO//EN" "gaigs_sho.dtd">

<show>
  <snap>
    <title>An example 3x2 array</title>
    <array>
      <row_label>Row 0</row_label>
      <row_label>Row 1</row_label>
      <row_label>Row 2</row_label>
      <column_label>Col0</column_label>
      <column_label>Col2</column_label>
      <column>
        <list_item>
          <label>[0][0]</label>
        </list_item>
        <list_item>
          <label>[1][0]</label>
        </list_item>
        <list_item color="red">
          <label>[2][0]</label>
        </list_item>
      </column>
      <column>
        <list_item>
          <label>[0][1]</label>
        </list_item>
        <list_item>
          <label>[1][1]</label>
        </list_item>
        <list_item>
          <label>[2][1]</label>
        </list_item>
      </column>
    </array>
  </snap>
</show>
```

And here is the one-snapshot show that GAIGS will produce from this code:



2.4.3 The bargraph structure type

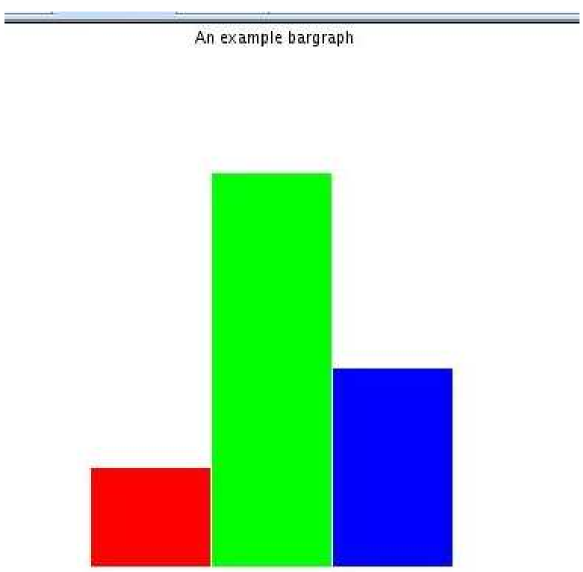
The bargraph is a simple structure:

```
<!ELEMENT bargraph (name?, bounds?, bar*)>

<!ELEMENT bar (label)>
<!ATTLIST bar magnitude CDATA #REQUIRED
              color CDATA "#000000">
```

After the standard name and bounds elements comes zero or more “bar” elements. These bar elements contain a label tag (which may be empty), and possess two attributes: a magnitude and a color. The range of (positive) values used for magnitudes does not matter as the structure will scale the height of the bars drawn to the screen accordingly – with the bar of greatest magnitude extending for the entire vertical bounds of the structure. The first bar given will be the one furthest on the left. Here is code for an example bargraph (with empty labels), followed by the image GAIGS produces when fed the code. If the labels are not empty, they appear as (potentially multi-line) labels centered under their respective bars.

This example uses the optional “bounds” element to resize the bargraph to insure that the largest bar does not extend for the entire vertical length of the snapshot's drawing window. The workings of the bounds element will be explained later.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE show PUBLIC "-//JHAVE//DTD GAIGS SHO//EN" "gaigs_sho.dtd">

<show>
  <snap>
    <title>An example bargraph</title>
    <bargraph>
      <bounds x1="0.2" y1="0.1" x2="0.8" y2="0.75"/>
      <bar magnitude="64"  color="red"><label></label></bar>
      <bar magnitude="255" color="green"><label></label></bar>
      <bar magnitude="128" color="blue"><label></label></bar>
    </bargraph>
  </snap>
</show>
```



2.4.4 The tree structure type

The tree structure comes in two varieties: general and binary. While a binary tree may be a special case of a general tree, at times it is be necessary to indicate visually whether a solitary child is the left or right child of its parent. Here is the base of the tree structure:

```
<!ELEMENT tree (name?, bounds?, (tree_node|binary_node)? )>
```

```
<!ATTLIST tree x_spacing CDATA "1.5"
               y_spacing CDATA "1.5">
```

After the standard name and bounds elements comes either a tree_node, which serves as the root of a general tree, or a binary_node, which serves as the root of a binary tree. The "tree" element also has two attributes, x_ and y_spacing. These control how far apart the nodes of the tree are spread in the x- and y-directions. The number is interpreted as a multiple of the diameter of the nodes (the nodes are sized to fit the text inside). The distance is measured from the center of each node, so the default values of 1.5 will result in the outer edges of the nodes being separated vertically and horizontally by one half of the nodes' diameters.

Here is the definition of a general tree's tree_node:

```
<!ELEMENT tree_node (label?, (tree_node,tree_edge?)* )>
<!ATTLIST tree_node color CDATA "white">
```

As you can see, each tree_node has a label (a #PCDATA element), and can have zero or more children tree_nodes, each followed by an optional tree_edge element describing how the parent is connected to the child coming immediately before the tree_edge. The tree_nodes also have a "color" attribute. The nodes are drawn on the tree from left to right.

The binary_node looks like this:

```
<!ELEMENT binary_node (label?, (left_node,tree_edge?)?, (right_node,tree_edge?)? )>
<!ATTLIST binary_node color CDATA "white">
```

The binary_node has a label, followed by zero, one, or two children nodes. The definitions for left_node and right_node are identical to the definition for the binary node. The difference in names only serves to identify a node as the left or right child of its parent. The optional tree_edges behave the same way in the binary tree as in the general tree: they describe the edge connecting the parent to the child that comes immediately before the description.

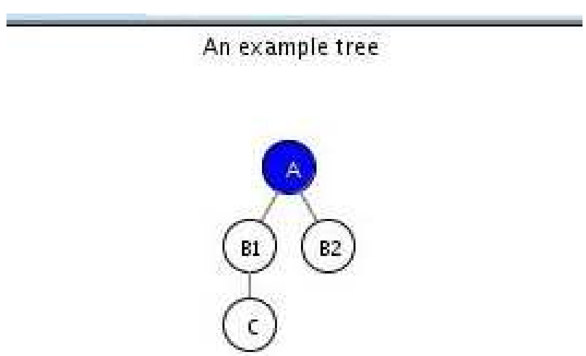
Here is the definition of the tree_edge element, used by both the tree_nodes and binary_nodes:

```
<!ELEMENT tree_edge (label?)>
<!ATTLIST tree_edge color CDATA "black">
```

The edges can be labeled (#PCDATA element) and/or colored.

Here is code for an example general tree, with one of the nodes highlighted light blue. The picture shows how GAIGS renders this.

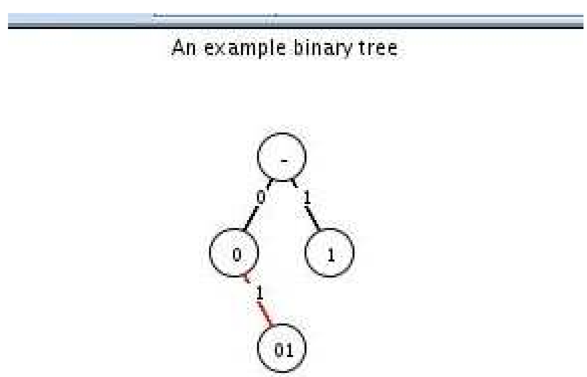
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE show PUBLIC "-//JHAVE//DTD GAIGS SHO//EN" "gaigs_sho.dtd">

<show>
  <snap>
    <title>An example tree</title>
    <tree>
      <tree_node color="light blue">     <!--  The root node labeled A-->
        <label>A</label>
        <tree_node>                      <!--  B1 is a child of A -->
          <label>B1</label>
          <tree_node>                    <!--  C is a child of B1 -->
            <label>C</label>
          </tree_node>
        </tree_node>
        <tree_node>                      <!--  B2 is a second child of A -->
          <label>B2</label>
        </tree_node>
      </tree_node>
    </tree>
  </snap>
</show>
```



Here is code for an example binary tree, with labeled edges and a colored edge. This code is also followed by an image.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE show PUBLIC "-//JHAVE//DTD GAIGS SHO//EN" "gaigs_sho.dtd">

<show>
  <snap>
```

```
    <title>An example binary tree</title>
    <tree x_spacing="2.0" y_spacing="2.0">
      <binary_node>                  <!--  A hyphen in the root -->
        <label>-</label>
        <left_node>                  <!--  0 the left child of the root -->
          <label>0</label>
          <right_node>               <!--  01 the right child of 0 -->
            <label>01</label>
          </right_node>
          <tree_edge color="red">
            <label>1</label>
          </tree_edge>
        </left_node>
        <tree_edge>
          <label>0</label>
        </tree_edge>
        <right_node>                 <!--  1 is right child of the root -->
          <label>1</label>
        </right_node>
        <tree_edge>
          <label>1</label>
        </tree_edge>
      </binary_node>
    </tree>
  </snap>
</show>
```



2.4.5 The graph structure type

The graph structure type can also be used to draw a network (that is, a graph with weighted edges). Here is the first part of the definition of a graph:

```
<!ELEMENT graph (name?, bounds?, vertex*)>
<!ATTLIST graph weighted (true|false) "false">
```

After the standard name and bounds elements, a graph has a set of zero or more vertex elements. A graph also has an attribute “weighted”. If “weighted” is set to “true”, the graph will load and draw edge weights.

Here is the definition of the vertex elements:

```
<!ELEMENT vertex (label?, position?, edge*)>
<!ATTLIST vertex color CDATA "white"
                 id CDATA #REQUIRED>
```

Each vertex can have a label (#PCDATA element). Then comes an optional position element, telling GAIGS where to draw the node. If the positions of the vertices are not specified, GAIGS arranges the vertices equally spaced around the circumference of a circle. This works well for small numbers of vertices, but for larger numbers it may be best to specify the vertices' positions using a graph vertex placement algorithm. GAIGS uses normalized screen coordinates to describe positions, so the coordinates should be between 0 and 1. Here is what the position element looks like:

```
<!ELEMENT position (EMPTY)>
<!ATTLIST position x CDATA #REQUIRED
                   y CDATA #REQUIRED>
```

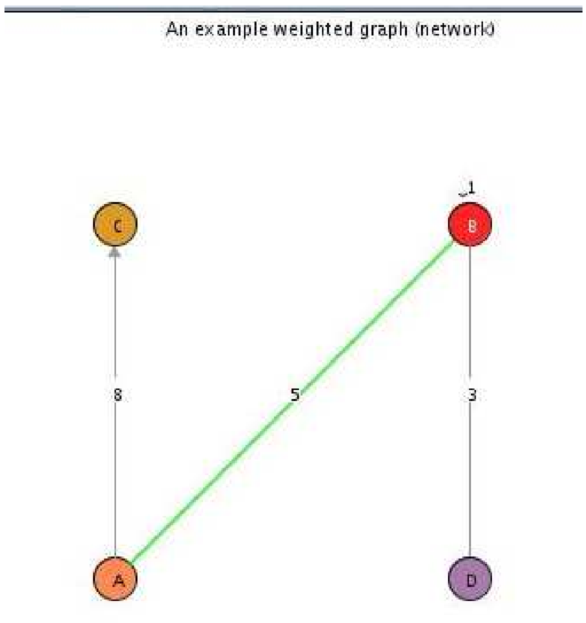
After the position element comes a set of zero or more edges. Each vertex has an “id” attribute that must be different from all other vertices' id's in the graph. The edge's “target” attribute should match the id of the vertex it connects to. Here is the definition of a vertex's edge:

```
<!ELEMENT edge (label?)>
<!-- target is matched with vertex.id: -->
<!ATTLIST edge target CDATA #REQUIRED
               directed (true|false) "false"
               color CDATA "#999999"> <!-- grey -->
```

If the “directed” attribute is set to true, an arrow is drawn on the edge pointing from the current vertex to the target. The edge can be labeled, if the graph's “weighted” attribute is set to true. Finally, the edge's color can be defined.

Here is the code for an example weighted graph (network), followed by an image of how GAIGS renders the code. (Note: As of August, 2005, the self-connecting edges are not being rendered properly, but luck this bug/feature should soon be resolved.)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE show PUBLIC "-//JHAVE//DTD GAIGS SHO//EN" "gaigs_sho.dtd">

<show>
  <snap>
    <title>An example weighted graph (network)</title>
    <graph weighted="true">
      <vertex color="#FF8855" id="A">
        <label>A</label>
        <position x="0.2" y="0.2"/>
        <edge target="B" color="#00FF00">  <!--  Bi-directional edge between A and B -->
          <label>5</label>
        </edge>
        <edge target="C" directed="true">  <!--  Directional edge from A to C -->
          <label>8</label>
        </edge>
      </vertex>
      <vertex color="#FF2222" id="B">
        <label>B</label>
        <position x="0.7" y="0.7"/>
        <edge target="B" color="red">     <!--  See Note above regarding self-connecting edges -->
          <label>1</label>
        </edge>
        <edge target="D">                 <!--  Bi-directional edge between B and D -->
          <label>3</label>
        </edge>
      </vertex>
      <vertex color="#DD9922" id="C">
        <label>C</label>
        <position x="0.2" y="0.7"/>
      </vertex>
      <vertex color="#AA77AA" id="D">
        <label>D</label>
        <position x="0.7" y="0.2"/>
      </vertex>
    </graph>
  </snap>
</show>
```

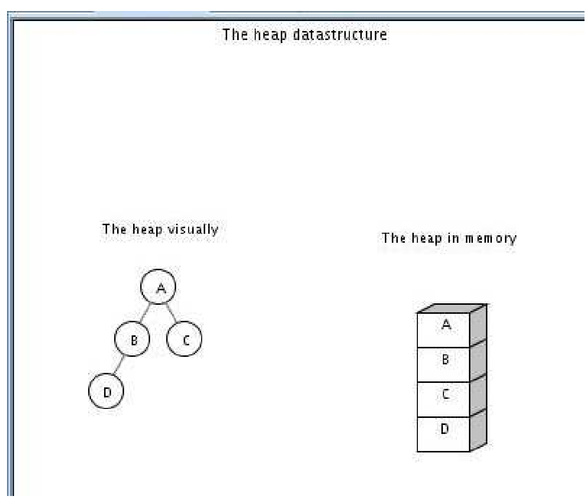


2.4.6 Drawing more than one structure to a snapshot

Here is the code for an example snapshot that resizes and positions two structures on the screen simultaneously:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE show PUBLIC "-//JHAVE//DTD GAIGS SHO//EN" "gaigs_sho.dtd">

<show>
  <snap>
    <title>The heap datastructure</title>
    <tree>                              <!--  First a tree view -->
      <name>The heap visually</name>
      <bounds x1="0.0" y1="0.2" x2="0.5" y2="0.7" fontsize="0.06"/>
      <binary_node>
        <label>A</label>
        <left_node>
          <label>B</label>
          <left_node>
            <label>D</label>
          </left_node>
        </left_node>
        <right_node>
          <label>C</label>
```

```
      </right_node>
    </binary_node>
  </tree>
  <array>                            <!--  Then the underlying array for the heap -->
    <name>The heap in memory</name>
    <bounds x1="0.5" y1="0.2" x2="1.0" y2="0.7" fontsize="0.06"/>
    <column>
      <list_item>
        <label>A</label>
      </list_item>
      <list_item>
        <label>B</label>
      </list_item>
      <list_item>
        <label>C</label>
      </list_item>
      <list_item>
        <label>D</label>
      </list_item>
    </column>
  </array>
</snap>
</show>
```



2.5 Interactive Questions

Each snapshot can have a question_ref element. A question_ref element comes at the end of the snapshot. It is empty and has an attribute “ref” whose value should match the “id” attribute of some question. The individual question elements (and their correct answers) come at the end of the "show" element, collected inside a "questions" element. The "questions" element is simply zero or more question elements:

```
<!ELEMENT questions (question*)>
```

Each question has two required attributes, the text of the question, and zero or more answer options:

```
<!ELEMENT question (question_text,answer_option*)>
<!ATTLIST question type CDATA #REQUIRED
                   id CDATA #REQUIRED>
  <!-- type: MCQUESTION|TFQUESTION|FIBQUESTION|MSQUESTION -->
```

The question's id should be unique from all other question id's and should match a snap's question_ref's "ref" attribute. The question's type must be one of the four question types provided, which are listed above in the comment: multiple-choice, true-or-false, fill-in-the-blank, or multiple-selection (check all of the possible answers which are correct). The question_text is simply #PCDATA:

```
<!ELEMENT question_text (#PCDATA)> <!-- the quesiton to ask the user -->
```

Here is the definition for the answer_option elements:

```
<!ELEMENT answer_option (#PCDATA)>
   <!-- TFQuestion: use text "true" or "false" (no quotes) for the correct answer -->
<!ATTLIST answer_option is_correct (yes|no) "no">
  <!-- specify "yes" only if it is a MC/MSQuestion. otherwise, ignored -->
```

For multiple-choice and multiple-selection questions, the answer_options listed in the question will be the possible choices to choose from when the question is asked. To define which is the correct answer, set the "is_correct" attribute of the answer_option to yes or no (a multiple-choice question should only have one correct answer). For fill-in-the-blank questions, simply provide in the answer_options' text acceptable answers (case-insensitive). For true-or-false questions, provide only one answer_option element with the text set to either "true" or "false" (no quotes). The MSQUESTION and MCQUESTION are the only types that care about the answer_options' "is_correct" attribute; both the FIBQUESTION and TFQUESTION ignore this attribute.

Here is the code for an example GAIGS show that asks a question of each of the four types. Each of the four snapshots contains nothing but a question_ref (and the required title). The structures would be added between the title and question_ref elements.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE show PUBLIC "-//JHAVE//DTD GAIGS SHO//EN" "gaigs_sho.dtd">

<show>

 <snap>
  <title>Question 1</title>
  <question_ref ref="k"/>
 </snap>

 <snap>
  <title>Question 2</title>
  <question_ref ref="222"/>
 </snap>

 <snap>
  <title>Question 3</title>
  <question_ref ref="abc"/>
 </snap>

 <snap>
  <title>Question 4</title>
  <question_ref ref="msq"/>
 </snap>

 <questions>
  <question type="TFQUESTION" id="k">
   <question_text>This statement is a question.</question_text>
   <answer_option>false</answer_option>
  </question>
  <question type="FIBQUESTION" id="222">
   <question_text>The answer to the ultimate question of life, the universe, and everything is:</q
   <answer_option>42</answer_option>
   <answer_option>forty-two</answer_option>
   <answer_option>pie</answer_option>
  </question>
  <question type="MCQUESTION" id="abc">
   <question_text>"McQuestion" sounds like an item on the menu of:</question_text>
   <answer_option is_correct="yes">McDonald's</answer_option>
   <answer_option is_correct="no">Hardee's</answer_option>
   <answer_option>Burger King</answer_option>
   <answer_option>Pizza Hut</answer_option>
  </question>
  <question type="MSQUESTION" id="msq">
   <question_text>"Which numbers are prime?"</question_text>
   <answer_option is_correct="no" >0</answer_option>
   <answer_option is_correct="no" >1</answer_option>
   <answer_option is_correct="yes">2</answer_option>
   <answer_option is_correct="yes">3</answer_option>
```

```
   <answer_option is_correct="no" >4</answer_option>
   <answer_option is_correct="yes">5</answer_option>
  </question>
 </questions>

</show>
```

The first question is true-or-false. The text of the question reads, "This statement is a question." The correct answer is specified between a set of answer_option tags: false.

The second question is fill-in-the-blank. The text of the question reads, "The answer to the ultimate question of life, the universe, and everything is:" The answers accepted as correct are "42", "forty-two", and "pie".

The third question is multiple-choice. The text of the question reads, "McQuestion" sounds like an item on the menu of:" Four answer options are provided, with the first being correct.

The fourth question is multiple-selection. The text of the question reads, "Which numbers are prime?" Six answer options are provided, with the third, fourth, and sixth being correct.

Chapter 3

Using the GAIGS support classes

The GAIGS support classes are a Java class library that provides extensive support for producing GAIGS XML snapshots by merely adding appropriate method calls to annotate the interesting events in your existing code. They essentially shield the programmer who is creating the visualization from having to know all the details about the XML that were described in the preceding chapter. The only price extracted from the programmer is that the data structures used in this code must come from the GAIGS data structure classes, each of which has a "toXML" method that knows how to create the necessary XML for that particular data structure. Producing a snapshot with one or more correctly positioned data structures is then as easy as calling the *writeSnapshot* method in the *ShowFile* class that is part of this library. Complete Javadocs for this library are available at *http://jhave.org/developer/doc/index.html*. Additional extensive slide sets describing its use can be found at *http://jhave.org/developer* – these slide sets have been used in numerous workshop we have presented on creating algorithm visualizations with the GAIGS support classes.

Chapter 4

Creating new GAIGS structures

4.1 Defining the XML

If you don't like GAIGS's built-ins, you do what every OO programmer does – extend and plug-in!

The first step is to decide on the XML for your extension. That will essentially define its syntax in a show. To illustrate, below we add a *foobar* element to the possible elements appearing in a <snap> tag:

```
<!ELEMENT show (snap+, questions?)>

<!-- Add other structure types to the (x|y|z)* part of snap -->
<!ELEMENT snap (title, doc_url?, pseudocode_url?, audio_text?
     (tree|array|graph|stack|queue|linkedlist|bargraph|node|foobar)*,
      question_ref?)>

<!ELEMENT title (#PCDATA)>
<!ELEMENT name  (#PCDATA)>
<!ELEMENT label (#PCDATA)>
<!ELEMENT doc_url (#PCDATA)>
<!ELEMENT pseudocode_url (#PCDATA)>

<!ELEMENT bounds (EMPTY)>
<!ATTLIST bounds x1 CDATA #REQUIRED
                 y1 CDATA #REQUIRED
                 x2 CDATA #REQUIRED
                 y2 CDATA #REQUIRED
                 fontsize CDATA "0.03">
```

```
<!-- Here we define foobar -->
<!ELEMENT foobar (name?, bounds?, nodelabel)>
<!ATTLIST foobar x CDATA "0.5"
               y CDATA "0.5"
               color CDATA "white">
<!ELEMENT nodelabel (EMPTY)>
<!ATTLIST nodelabel text CDATA "">
```

The rest of the DTD remains unchanged. For example, consider a <show> adhering to this DTD and defining a “foobar”.

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- <!DOCTYPE show PUBLIC "-//JHAVE//DTD GAIGS SHO//EN" "gaigs_sho.dtd"> -->
<!-- Use local SYSTEM DTD instead of the PUBLIC DTD -->

<!DOCTYPE show SYSTEM "gaigs_sho.dtd">

<show>
  <snap>
    <title>An example of a simple foobar</title>
    <foobar x="0.25" y="0.25" color="#FF0000">
      <name>foobar1</name>
      <bounds x1="0.2" y1="0.1" x2="0.8" y2="0.75"  fontsize="0.1"/>
      <nodelabel text="The first foobar"/>
    </foobar>
  </snap>

  <snap>
    <title>An example of two simple foobars</title>
    <foobar x="0.25" y="0.25" color="#0000FF">
      <name>foobar2</name>
      <bounds x1="0.0" y1="0.0" x2="0.45" y2="0.45"  fontsize="0.1"/>
      <nodelabel text="The second"/>
    </foobar>
    <foobar x="0.25" y="0.25" color="#00FF00">
      <name>foobar3</name>
      <bounds x1="0.45" y1="0.45" x2="0.9" y2="0.9" fontsize="0.1"/>
      <nodelabel text="The third"/>
    </foobar>
  </snap>

</show>
```

In the source code for GAIGS, every data structure has *StructureType* as an ancestor.

```
StructureType
```

```
|-- Md_Array
|-- LinearList
    |-- Stack
    |-- Queue
    |-- LinkedList
    |-- Bar
|-- BinaryTree
|-- GeneralTree
|-- Graph_Network
    |-- Ggraph
    |-- Network
```

Hence coding your own structure type in Java source code is as simple as writing one file, the *YourStructureName.java* file containing the class *YourStructureName*, which inherits at some point from the abstract class *StructureType*. Both of these files will be located in the *jhave2/gaigs/src/gaigs2/* directory, which can be built by running ant from the *jhave2/gaigs/* directory. The StructureType class is an abstract base, as indicated in the following documentation:

```
// All GAIGS Structures inherit from ...
abstract class StructureType {

    // StructureType has only one parameterless constructor.

    // All derived structures should provide their own parameterless
    // constructor and override the following two methods:

    // Load the structure from the root of its XML tree (JDOM style)
    void loadStructure(Element myRoot, LinkedList llist, draw d)   Parse the script

    // All derived StructureTypes should override this -- be sure to
    // call super on this method when your structure is empty
    void drawStructure (LinkedList llist, draw d)      And render it ...

    // *** USEFUL PROTECTED METHODS AND VARIABLES

    // Access your GKS graphics routines thru this. Send drawing commands
    // to this Localized GKS object in normalized [0,1]x[0,1] coordinates
    // describing your position within the bounds given to LGKS
    protected LocalizedGKS LGKS;

    // For derived objects who want to know their bounds
    protected double structure_fontsize, structure_left,
                     structure_right, structure_bottom, structure_top;

    // Load name and bounds info common to all localized derived structures
    public void load_name_and_bounds(Element my_root, LinkedList llist, draw d)

    // Establish the protected variables that determine the starting vertical
    // coordinate of the title (and its ending vertical coordinate).
    public void calcDimsAndStartPts(LinkedList llist, draw d)

    // Given s, return its normalized width  Often used to size a box or circle around text
```

```
protected double normalized_width(String s)

// Given color as a (usually hex) string, convert it to the right
// Java color as int
protected int colorStringToInt(String color)

// Given a fill-area color as a (usually hex) string, convert text
// to appear in the fill area to the right Java color as int
protected int colorStringToTextColorInt(String color)
```

4.2 Your structure's LGKS object is what you "draw" with.

```
// YOUR LGKS OBJECT RESPONDS TO THE FOLLOWING MESSAGES
// The LinkedList and draw objects always tag along ...

// Set the interior style for a fill area.  The int color is
// typically obtained from the hex string by StructureType's
// colorStringToInt method.  Sorry, only the constants bsClear and
// bsSolid are presently supported for style.  And, of course, a
// clear fill area is just ...
public void set_fill_int_style(int style, int color, LinkedList seginfo, draw d)

// Draw a fill area with the specified number of points and their
// coordinates
public void fill_area(int numpts, double ptsx[], double ptsy[],
                      LinkedList seginfo, draw d)

// Draw a polyline with the specified number of points and their
// coordinates
public void polyline(int numpts, double ptsx[], double ptsy[],
                     LinkedList seginfo, draw d)

// Set the text alignment.  Choices for horiz and vert are:
//     final static int TA_CENTER    = 0;
//     final static int TA_LEFT      = 1;
//     final static int TA_RIGHT     = 2;
//
//     final static int TA_BASELINE  = 0;
//     final static int TA_BOTTOM    = 1;
//     final static int TA_TOP       = 2;
public void set_text_align(int horiz, int vert, LinkedList seginfo, draw d)

// Set the color (as a Java int) for drawing text.  Usually this
// int is obtained from the hex string by your having called
// colorStringToTextColorInt in StructureType.java
public void set_textline_color(int color, LinkedList seginfo, draw d)

// Change the font size
public void set_text_height(double height, LinkedList seginfo, draw d)

// Draw your text at the specified coordinate
public void text(double x,double y, String str, LinkedList seginfo, draw d)

// Draw an ellipse from start angle thru end angle
public void ellipse(double x, double y, double stangle, double endangle,
                    double xradius, double yradius,
                    LinkedList seginfo, draw d)
```

```
// Draw a (outlined) circle
public void circle(double x, double y, double radius, LinkedList seginfo, draw d)

// Draw a filled circle
public void circle_fill(double x, double y, double radius,
                        LinkedList seginfo, draw d)
```

4.3 Use the JDOM XML (www.jdom.org) parsing class to write loadStructure

The JDOM essentials you will need ...

- The Element class provides the type of nodes in the XML tree, e.g.,

  ```
  public void loadStructure(Element rootEl, LinkedList thingsToRender, draw drawerObj)
  ```

- Given an Element, you can use *getChild* to get the first child and *getChildren* to return a *List* of children
- *getText* returns the text (as a String) of a node in the tree
- *getAttributeValue(String which-attrib)* return the value of a particular attribute

Here's the resulting *foobar.java* file

```
// foobar.java
// A sample (and simple) extension of the GAIGS StructureType

package gaigs2;
import java.awt.image.*;
import java.util.*;
import org.jdom.*;

public class foobar extends StructureType {

    double circle_center_x, circle_center_y; // center coords
    double circle_rad;          // radius

    int circle_color;           // node color

    int circle_labelcolor;      // text color
    String circle_label;        // only set up for a single line of text

    // Must provide a parameterless constructor for instantiation via reflection
    public foobar() {
        super();                // necessary

        circle_color = White;   // our hex notation is "#RRGGBB"
        circle_labelcolor = Black;
        circle_label = null;
        circle_center_x = 0.50;
        circle_center_y = 0.50;
        circle_rad = 0.25;
    } // foobar()
```

```
// This initialization method gets passed a jdom.Element whose
// name is "foobar". So gaigs_sho.dtd must be modified, adding a
// "foobar" element to the list of structure types a snap can
// contain.
public void loadStructure(Element rootEl, LinkedList thingsToRender, draw drawerObj) {
    // This call loads the name and bounds if your xml
    // structure-element has a name and/or bounds like the
    // built-in structures.
    load_name_and_bounds(rootEl, thingsToRender, drawerObj);
    calcDimsAndStartPts(thingsToRender, drawerObj);

    // JDOM, AT LEAST AS MUCH AS WE NEED IT, IS EASY TO USE
    List children = rootEl.getChildren(); // getChildren returns a list
    Iterator iter = children.iterator(); // which we will iterate through

    Element labelEl;

    // NOTE: This is only an unnecessary illustrative loop, since
    // we could get what we want directly
    while( iter.hasNext() ) {
        Element child = (Element) iter.next(); // walk through the list of children

        if( child.getName().equals("name") ) {
            // Just showing we could get it if we wanted,
            //  but already done for us in load_name_and_bounds(..)
            String junkName;
            junkName = child.getText(); // get the text of this node in the XML tree
        }
        else if( child.getName().equals("bounds") ) {
            //  Just showing we could get it if we wanted,
            //  but already done for us in load_name_and_bounds(..)
            double junkBound;
            junkBound = Format.atof( child.getAttributeValue("x1") ); // get an attrib
        }
        else if( child.getName().equals("nodelabel") )
            labelEl = child;
    } // End illustrative loop

    // In this example, we could get the element we want directly --
    labelEl = rootEl.getChild("nodelabel");

    // The XML is validated against the DTD, so if there is a
    // #REQUIRED attribute or a default value we can safely assume
    // it is there
    circle_center_x = Format.atof( rootEl.getAttributeValue("x") );
    circle_center_y = Format.atof( rootEl.getAttributeValue("y") );

    circle_color = colorStringToInt( rootEl.getAttributeValue("color") );

    if( labelEl != null ) {
        circle_label = labelEl.getAttributeValue("text");
        circle_labelcolor = colorStringToTextColorInt( rootEl.getAttributeValue("color"
        circle_rad = normalized_width(circle_label)  / 2.0;
    }
} // loadStructure


// Use the LGKS object to draw the structure
public void drawStructure(LinkedList thingsToRender, draw drawerObj) {
    // draw the circle(filled)
    LGKS.set_fill_int_style(bsSolid, circle_color, thingsToRender, drawerObj);
    LGKS.circle_fill(circle_center_x, circle_center_y, circle_rad, thingsToRender, draw
```

```
        // draw the circle outline
        LGKS.set_textline_color(Black, thingsToRender, drawerObj);
        LGKS.circle(circle_center_x, circle_center_y, circle_rad, thingsToRender, drawerObj);

        // draw the label
        LGKS.set_textline_color(circle_labelcolor, thingsToRender, drawerObj);
        LGKS.set_text_align(TA_CENTER, TA_BOTTOM, thingsToRender, drawerObj);
        LGKS.text(circle_center_x, circle_center_y, circle_label, thingsToRender, drawerObj);
    } // drawStructure

} // class foobar
```

Chapter 5

Pseudo-code, Audio Support, and Input Generators

5.1 Adding synchronized pseudocode to the snapshots in your script

Presently the technique we use to have the JHAVÉ pseudocode window synchronize with the snapshots in your visualization script is based on PHP pages that reside on the JHAVÉ server. These PHP scripts read an XML file that contains your pseudocode and, based on the parameters received by the PHP script, the appropriate synchronization highlighting is done. A set of PDF slides that explains the underlying XML and how to use it to generate the synchronized pseudocode is online at:

http://jhave.org/developer/documentation_and_slides/using_pseudocode.pdf

5.2 Adding audio explanations to the snapshots in your visualization script

The XML and GAIGS support classes have recently been extended to allow each snapshot to be accompanied by an audio explanation. This audio explanation may be in the form of a *.wav* or *.au* file, or it may use the text-to-speech library from FreeTTS (*http://freetts.sourceforge.net/docs/index.php*). Details about how to add audio support to your snapshots is available online at:

http://jhave.org/developer/documentation_and_slides/audio.pdf

5.3 Having students generate input data sets for the algorithm

An engagement technique that will help students learn from watching visualizations is to have them design input data sets for the algorithm that will consequently generate an execution of the algorithm meeting a certain set of requirements or constraints. Although your script-producing program ultimately takes its input as command-line parameters, when students launch that program by connecting to the JHAVÉ server, they will see a pop-up GUI called an "input generator" to get that input from them. The data from that input generator is then transmitted back to the server (in XML form) where it is processed by a front-end program that parses the XML into the command-line parameters that your script-producing program ultimately uses when it runs. Details on writing an input generator for your script-producing program can be found online at:

http://jhave.org/developer/documentation_and_slides/input_generators.pdf

Chapter 6

Questions

6.1 Adding questions to a script file from a program that is writing the script file

(Note: If you are using the new GAIGS support classes described in Chapter 3, you will not need to know the details described in this chapter. The support classes will handle everything for you in terms of splicing your questions into the XML visualization script.)

- Instantiate a questionCollection – essentially a Vector of questions with a few additional special operations

```
public class questionCollection {

    // Constructor with the output stream to write questions to
    public questionCollection(PrintWriter out){

    // Add question q to the collection
    public void addQuestion(question q){

    // Write the tag for question at index into the output stream
    public void insertQuestion(int index){

    // Write the text of all questions and answers at the end of the script
    public void writeQuestionsAtEOSF(){
}
```

- At times where your script writing program wants to ask a question, instantiate a *tfQuestion*, *mcQuestion*, or *fibQuestion*, all of which extend from the abstract *question* class:

```
public abstract class question{

    // Constructor
    public question()

    // questionText is a string containing the text for this question
    public void setQuestionText(String questionText){
}
```

- Each specific derived question has a constructor that accepts a string id/tag for the question and a *setAnswer* method use to establish the answer for this question. For example the *fibQuestion* class:

```
public class fibQuestion extends question{

    // Construct the fib question, providing its identifying string
    public fibQuestion(PrintWriter out, String id){

    // Set the answer for this fib question
    // Use \n to separate different answers that are allowed
    public void setAnswer(String answer){
```

- So, the general algorithm to create questions is (see *LinearHashing.java* for complete program containing this algorithm:

```
for each snapshot you create
  If you want a question with this snapshot
     Manufacture text of question and the answer
     fibQuestion quest = new mcQuestion(out, (new Integer(qIndex)).toString());
     qIndex++;  // Increment your question counter
     quest.setQuestionText(string-containing-question);
     quest.setAnswer(the-answer);
     Questions.addQuestion(quest);
     Questions.insertQuestion(qIndex);
  Now write the snapshot that is associated with the question
```

- After all the snapshots have been written, be sure to:

```
Questions.writeQuestionsAtEOSF();
```

Chapter 7

Writing a JHAVÉ Visualizer

7.1 Extending Visualizer

The Visualizer class was designed to be an abstract layer between the classes that implement a visualizer's behavior and the client. The class is as an adapter, transforming the API of the individual visualizers into a uniform API the client knows how to handle.

The Visualizer class is an abstract class that has several convenience methods built in to facilitate development. The class handles listeners and facilitates firing events to the listeners.

7.1.1 Packages of interest

jhave.core This package contains the Visualizer class and, in the future, it will contain other classes that are critical to JHAVÉs infrastructure.

jhave.event This package contains all of the Events and EventListeners that are used by the Visualizer class to notify the client of changes within the visualizer. The Visualizer developer should have no need to use the EventListeners defined in the package, but can choose to use the event classes directly when firing events to the client.

jhave.question This package contains the revised Question System created for JHAVÉ. The primary classes to deal with are Question, QuestionFactory, and QuestionParseException. For more information check the JavaDocs on these classes.

7.1.2 Visualizer Class

The Visualizer class is an abstract class which has only three abstract methods (at the moment) with eight control methods that are overloaded if the Visualizer supports that particular operation (such as play).

7.1.3 Construction

A Visualizer is only considered legal if it has a constructor with a single parameter of a *java.io.InputStream*. The reason for this is due to how Visualizers are to be instantiated by the client. To satisfy the compiler the first call in this constructor should be
super(the inputstream).

The constructor also needs to set the visualizers capabilities by calling *setCapabilities()* that is contained in the parent class. By default the capabilities are set so that the visualizer is not controllable.

A capabilities mask can be created by adding the several CAP_* constants that are provided in the Visualizer class. For example this would set the visualizer as being controllable and supporting the play operation:

```
setCapabilities(CAP_CONTROLLABLE + CAP_PLAY);

For more info on these see the Visualizer class JavaDoc.

The following is an example constructor:

public class TestVisualizer {
        public TestVisualizer(InputStream script) {
                super(script);

                // Parse the script

                        setCapabilities(CAP_CONTROLLABLE + CAP_PLAY);
}
}
```

7.1.4 Abstract Methods

getCurrentFrame() Returns the current frame or key frame (zero based). This will return 0 if we are at the beginning of the visualization and a value equal to
(getFrameCount() 1).

getFrameCount() Returns the number of frames or key frames. Must act the same as how length works on arrays or size() work on collections.

getRenderPane() Returns the component in which the animation or visualization is displayed.

7.1.5 Control Methods

Fuller descriptions of these can be found in the Visualizer class JavaDoc. Here we only list some of the potential caveats of the methods.

play() Play should be blocking, meaning that it does all of its work before exiting the method. This is the same way that a modal dialog doesnt return (from a setVisible(true) call) until the dialog has been disposed. Play is only available in the controls if CAP_PLAY is set.

pause() Pause should be written so that it causes play to exit immediately (this could be between frames / key frames in an animation). The visualizer should be written so it can gracefully recover from a paused animation that is not at a key frame. There is some thought that if play were to continue until finishing animating to the next frame that would be OK as well, but this has yet to be tested. Pause is only available in the controls if CAP_PLAY and CAP_PAUSE are set.

stop() Stop should pause the animation (if it is animating) and return to the first frame. If it is not animating, then it should return to the first frame. Stop is only available in the controls if CAP_PLAY and CAP_STOP are set.

stepBackward() As with play, stepBackward needs to block until after animation or switching to the previous frame is done. Step backward is only available if CAP_STEP_BACKWARD is set.

stepForward() stepForward will advance one frame / key frame forward, and should block as with play and stepBackward. stepForward is only available if CAP_STEP_FORWARD is set.

gotoFrame(int frame) Goto frame should skip directly to the given frame or key frame without doing any animation. The controls are currently setup to make a call to gotoFrame when the slider control is moved. So moving the slider from frame 8 to frame 1 will display all the frames between 8 and 1 as well. This has the possible caveat of also displaying the question associated with that frame for all of those frames. The frame slider is only available when CAP_GOTO_FRAME is set.

zoom(double level) Change the magnification (scale) of the image / animation. Value of 1 is 100% and .25 is 25%. Right now the controls are setup so that zooming in and out change the zoom by .25. (I havent done much work with this yet outside of GAIGS, but I know the size of the rendering pane will have to change its size (setPreferredSize()) to match the new size of the image so that scrolling works correctly. I believe a call of revalidate() will be needed on the rendering pane.) Zooming controls are only visible when CAP_ZOOM is set.

double getZoom() Returns the current zoom level. No need to worry about this unless CAP_ZOOM is set.

7.2 Events and Listeners

This is how the Visualizer alerts EventListeners of question, documentation, and audio changes. Fortunately due to the hierarchical nature of Visualizer implementations, the listeners and event firing can be handled in the Visualizer class with convenience methods so that visualizers dont need to deal directly with events (except DocumentEvents to a small degree where the type of the event must be passed as a parameter).

To this end there are three convenience methods for firing QuestionEvents, DocumentEvents, and AudioText events respoectively.

fireQuestionEvent(Question q) Calling this method causes a new QuestionEvent to be created with the Question and sent to all of the QuestionListeners.

fireDocumentationEvent(URI document, int type) The document is given as a java.net.URI and is given in the form: `[scheme:][/path/to/file]`

The schemes that are available are:

Scheme	Description	Example
http	General HTTP URI / URL, should point to an exact address	http://www.google.com/index.html
res	Used to load a file that is on the classpath. It is loaded by using the system classloader to obtain a URL	res:org/gaffneyc/image1.jpg
rel	Used to load a file that is relative to the Webroot. This is also translated to use the algorithms name to locate the exact location of the file.	rel:kruskalrichard.html

There are other schemes available, these are the default URL types supported by the java virtual machine. The resource (res) and relative(rel) are custom defined types and are case-insensitive.

fireAudioTextEvent(String the_text) The string to be spoken. If the string contains '.au' or '.wav', the assumption is made that this is a url for an audio file. Otherwise the speech-to-text module is used to speak the text.