**Virginia Polytechnic Institute and State University**

**Virtual Textbook Developer Guide**

**Modeling a concept graph based statics textbook**

*Last Updated: July 9, 2012*

Table of Contents

[I.Background 3](#__RefHeading__996_1238377174)

[ZVTM 3](#__RefHeading__998_1238377174)

[Content 3](#__RefHeading__1000_1238377174)

[II.Basic Class Layout 4](#__RefHeading__1002_1238377174)

[III.Other Useful Information 4](#__RefHeading__1004_1238377174)

[Node Focus Algorithm 4](#__RefHeading__1006_1238377174)

[Realigning Node Information 5](#__RefHeading__1008_1238377174)

[Drawing Node Title and Description Text 5](#__RefHeading__1010_1238377174)

[Navigating to Web Pages ( Chapter level view, specifically) 6](#__RefHeading__1012_1238377174)

[Subnodes 6](#__RefHeading__1080_1238377174)

[Generating GraphViz data 6](#__RefHeading__1060_1238377174)

[HTML Code to prevent browser window scrolling when zooming 6](#__RefHeading__1062_1238377174)

[IV.Uploading the Applet and Server Setup 7](#__RefHeading__1014_1238377174)

[First-Time server Setup 7](#__RefHeading__1064_1238377174)

[V.Changes 8](#__RefHeading__1016_1238377174)

# Background

## ZVTM

The Virtual Textbook Application (VTA) is built using a graphics and animation toolkit called ZVTM (<http://zvtm.sourceforge.net/>). Although the ZVTM Developer’s Guide can be useful, it is merely a document that should be looked over and not referenced frequently; all of the basics that it describes from ZVTM have been implemented and wrapped in wrapper classes in the implementation of the VTA. ZVTM is mostly being used to help support the animation of the application, the graphics, which will further be referenced by the proper ZVTM name for the graphical components, glyphs, and the features of having a “world” that is zoomable and navigable.

To wrap the ZVTM components, we used the model.Node, model.Link, controller.VirtualTextbookApplet, and view.AppCanvas classes, as well as an implementation of the ZVTM listener interface ViewEventHandler in view.CameraMovementListener. The Node class utilizes most of the glyphs that are seen in the application. Although this class also contains all of the nodes’ information, it also contains a VText for the title, a VText for the description, and a VRectangle for the surrounding rectangle for each node. The AppCanvas class contains most of the camera variables such as the vSpaceManager, which is the all-encompassing and variable that stores almost all of the camera, views, perspectives, etc. for ZVTM, the detailedSpace, which is the space that the glyphs of the application reside in, and the detailedCamera, which is the actual camera of the application that is allowed to change to get different views of the detailedSpace. The Link class extends the VSegment class, which draws line glyphs, to implement the links from node to node in the application; the attributes of each link is stored within an instantiation of the Link class’ inner class LinkProperties. Finally, the VirtualTextbookApplet is a JApplet class that sets up the application to be run as an applet, as well as provides the Container for the application to be put in.

## Content

The content of the application is stored as an XML document within the src/content folder. This content is easily understood, but contains linktypes to declare what links are available and what attributes, such as name and color, each link type has. The content document also has chaptertypes which similarly declare which chapters are available and what attribute each chapter should have. Finally, the bulk of the content document is the actual node information that contains all of the details for each node. The document is parsed in the controller.XmlParser class in the following order, as can be seen in view.AppCanvas.populateCanvas():

1. chaptertypes
2. nodes
3. linktypes
4. links

# Basic Class Layout

The VTA is laid out with the goal of utilizing a Model-View-Controller (MVC) design pattern. This has separated the project into three main packages. Below is the package hierarchy to supplement further explanation of the class layout. (Note: bold indicates a package, not a class)

|  |  |  |
| --- | --- | --- |
| * **controller**   + Configuration   + Main   + VirtualTextbookApplet * **controller.xml**   + - GraphViz     - XmlParser | * **model**   + Link   + Node   + NodeMap | * **view**   + AppCanvas   + CameraMovementListener |

The controller package contains most of the setup methods to initialize an instantiation of AppCanvas with variables depending on if the application is being run as an applet or as a standalone application. The model package contains the bulk of the information for the application, including a static reference to all of the nodes (in Node class), as well as a static reference to all of the links (in Link class). The model package also has the wrappers for the ZVTM glyphs, which would be better put in the view class if they were planning to be changed or if it could be decoupled from the node and link information. Finally, the view class contains the listener class for the application as well as the canvas (in ZVTM terms, the cameras and views) that handles and displays the graphics for the application.

# Other Useful Information

## Node Focus Algorithm

Node positioning is handled by the NodeMap class, specifically through the Graphviz related functions. See the section of Graphviz data for more information on how this is created. The CameraMovementListener handles catching user interaction, and sends calls to AppCanvas, which then determines which view needs to be shown (chapter or overview), shows/hides nodes, and calls the layout function in NodeMap through the method showSelectedChapter(). Of note is that CameraMovementListener keeps track of the currently highlighted node, not AppCanvas.

## Realigning Node Information

Although the glyphs that make up the nodes are “stuck” together using ZVTM’s Glyph.stick() method, the actual rectangle and the positioning of the glyphs is updated dynamically, when necessary using Node.realignNodeText(), which is called using draw whenever the graphics are updated and whenever a move event is called for the node. As draw events are called very frequently, a performance boost might be achieveable if this method is only called when the node is moved, and might be worth looking into if runtime performance becomes too slow. This is not entirely straightforward though, as simply removing the call from draw and only having the call in all node movement methods does not correcly refresh the text.

## Drawing Node Title and Description Text

So for us to handle creating multiple lines of text for a node's title and description, we had to edit some ZVTM code. All the edited code is in VText.java located at "fr\inria\zvtm\glyphs\VText.java". Initially, only one line of text would be created. ZVTM uses the drawString() of AWT. On lines 427 and 430 we replaced the call to drawString() with a call to our method drawTheString().

The drawTheString() method will take the VText protected String text and split it into an array of the words. The AWT drawString() method will be called for each row of words. The number of words in a row is determined by the number of characters in each words, and respects the static variable MAX\_CHARS\_PER\_LINE. This method is also where the textContainerHeight and textContainerWidth are determined based on the longest row and the number of rows. These variables are also new additions to the original ZVTM code. Note that the minimum node width set in Configuration.java is always respected despite this calculation.

**drawTheString()** works as follows:

1. The string is split into an array of words
2. If the number of total characters is greater than the max characters per row (MAX\_CHARS\_PER\_LINE), then multiple rows need to be drawn
   1. While the endIndex is not at the end of the words array
      1. If the current index is the last index, draw the all the words from the last line to the current index, and break
      2. Otherwise if the number of total characters from the words at startIndex to the word at endIndex is less than MAX\_CHARS\_PER\_LINE, add the number of characters from the word at endIndex to the current character count.
      3. Otherwise, draw the words from startIndex to endIndex in the current row, and move to the next row, moving startIndex and endIndex to the next unused word.
   2. After the loop, the textContainerHeight is determined by the height of the characters multiplied by the number of rows
3. Otherwise just draw the string

## Navigating to Web Pages ( Chapter level view, specifically)

The navigation to web pages is handled by the AppCanvas.navigateTo() method, and is primarily called by the CameraMovementListener.click1() method when a selected node is clicked. Webpages are addressed relative to the application folder on the server, and by convention are kept in the webpages folder.

## Subnodes

Subnodes are nodes that represent many nodes with the same link type grouped together. These are created only for nodes not in the current node's chapter, and only if there are 3 or more nodes to group. These nodes are treated as part of a regular node, and code relating to them can be found in Node.java.

## Generating GraphViz data

Graphviz data is generated by running the application as a standard java application. To do this export as 'Runnable JAR file', and run the application. The graphviz data will be created in the directory where you run the jar. NOTE: Graphviz must be installed, the location is determined in the GraphViz.java file:

// private static String TEMP\_DIR = "/tmp"; // Linux

**private** **static** String *TEMP\_DIR* = "c:/temp"; // Windows

// Where is your dot program located? It will be called externally.

// private static String DOT = "/usr/bin/dot"; // Linux

**private** **static** String *DOT* = "C:/Program Files (x86)/Graphviz 2.28/bin/dot.exe"; // Windows

Make sure the temp directory exists, or graphviz will not be able to execute. The generated files are then placed in the content folder on the server.

## HTML Code to prevent browser window scrolling when zooming

This code should be placed in the html below the applet tag, and the applet should have the id='Adaptive Map' and name='Adaptive Map' parameters.

*<script type="text/javascript">*

*window.onload = function() {*

*var applet = document.getElementById("Adaptive Map");*

*applet.onmouseover = function(e) {*

*window.onscroll = function(e) {*

*//scroll where you want to be (the top)*

*scroll(0,0) ; }*

*}*

*applet.onmouseout = function(e) {*

*//replace with empty function*

*window.onscroll = function(e) {} }*

*};*

*</script>*

# Uploading the Applet and Server Setup

## First-Time server Setup

To run the applet on the server you need java and tomcat installed and running. The files that go on the server can be found in the Tomcat Supplementary Folder. Create a new folder called AdaptiveMap in the Tomcat 7.0/webapps folder, and copy all the files over. When compiling the new jar file, export as 'JAR file', save it as adaptive-map.jar and tick the following options:

* Export generated class files and resources
* Export Java source files and resources
* Compress the contents of the JAR file

To get the custom browser icon to work, you must also copy it to the webapps/ROOT folder.

The index.html file should also be editted so that the icon shows up and users are correctly redirected to the applet. Add:

<head>

<meta http-equiv="refresh" content="0;URL=http://adaptivemap.me.vt.edu/AdaptiveMap/index.html">

</head>

<link rel="icon"

type="image/ico"

href="icon/favicon.ico">

to the index.html file located in the ROOT folder.

# Changes

**(Between December 13, 2011 and May 3, 2012)**

1. General Changes
   1. Set up a github repository for version control (<https://github.com/mpascale/Adaptive-Map>)
   2. Set up server on ME machine to run the Adaptive Map (<http://128.173.188.251:8080/>)
   3. Created NodeMap data structure to store nodes according to chapters
   4. Made outer view accessible through right-clicking
   5. Made mid-view to overview transition clean (mid-level nodes move back to their original positions and disappear)
   6. Changed all textboxes to determine line length based on characters
   7. Added switching between overview/chapter view, remembering last selected node
   8. Set navigateTo(url) to use relative addressing scheme.
   9. Updated all comments and documentation
2. Changes to Overview
   1. Added zoom/search buttons and legend
   2. Centered nodes at start
   3. Hide individual nodes when in overview
   4. Increase node size and prevent overlap
   5. Make strength of node gradient dependent upon number of nodes in chapter
   6. Do not allow chapter nodes to change view (except to become hidden)
   7. Realigned text
   8. Reversed order of buttons- Book/Chapter/Page
   9. Made background color based on current chapter color
   10. Links between chapters have weight dependent on # of links between nodes
3. Graphing Changes
   1. Implemented graphing algorithm to place nodes in optimal positions so links do not cross as often
   2. Added 1-level out nodes to the mid-level view (placed them in empty spots left from the graph layout)
4. Implemented Search Function
   1. Search any node in the map from the chapter view
   2. For searches that produce multiple results, allow user to select desired node from a pop up list
5. Changes to Links
   1. Made links highlight and display their link type when connecting nodes are hovered over
   2. Changed link colors to be more visually appealing
   3. Fixed ghost links
6. Changes to Nodes
   1. Color scheme
   2. Gradient colors
   3. Rounded edges
   4. Bolded titles/ non bold descriptions
   5. Created constant node size based off size of largest node text for the chapter