**Virginia Polytechnic Institute and State University**

**Virtual Textbook Developer Guide**

**Modeling a concept graph based statics textbook**

*Last Updated: May 3, 2012*

Table of Contents

[I. Background 1](#_Toc311624971)

[ZVTM 1](#_Toc311624972)

[Content 1](#_Toc311624973)

[II. Basic Class Layout 2](#_Toc311624974)

[III. Other Useful Information 2](#_Toc311624975)

[Node Focus Algorithm 2](#_Toc311624976)

[Realigning Node Information 4](#_Toc311624977)

[Drawing Node Title and Description Text 4](#_Toc311624978)

[Navigating to Web Pages (zoom level-3, specifically) 4](#_Toc311624979)

[IV. Uploading Applet 4](#_Toc311624980)

# 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. This is different than when the application runs as a standalone application and Main.main is executed, which creates a JFrame to put the application 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**     - GridsGenerator     - 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

The code for the algorithm that handles the rearranging and appearing and disappearing of nodes and links when a node is focused on can be found in CameraMovementListener.click1(). The bulk of the actual algorithm for determining where nodes are going to be positioned however, can be found in Node.getGridLocations(), as well as pasted below.

// Find all of the nodes that will go in the grid

List<Node> nodesToShow = **new** ArrayList<Node>();

// Get the nodes one level out

List<Link> selectedNodeLinks = selectedNode.getNodeLinks();

**for** (Link firstNodeLink : selectedNodeLinks) {

Node firstLevelNode = firstNodeLink.getFromNode() == selectedNode

? firstNodeLink.getToNode()

: firstNodeLink.getFromNode();

nodesToShow.add(firstLevelNode);

// Get the nodes two levels out

List<Link> secondLevelLinks = firstLevelNode.getNodeLinks();

**for** (Link secondLevelLink : secondLevelLinks) {

Node secondLevelNode = firstLevelNode == firstNodeLink

.getFromNode()

? secondLevelLink.getToNode()

: secondLevelLink.getFromNode();

nodesToShow.add(secondLevelNode);

}

}

// Find where the nodes will go in the grid

Map<Node, GridLocation> gridLocations = **new** HashMap<Node, GridLocation>();

// Give the middle 1 to account for the center node

**int**[] rowCount = {0, 0, 1, 0, 0};

**for** (Node node : nodesToShow) {

**if** (node.equals(selectedNode) || gridLocations.containsKey(node)) {

**continue**;

}

**int** y = *findDepthBetween*(selectedNode, node);

// Add 2 to since depth can be from -2 to +2, and the array is

// indexed from 0 - 4

**if** (y + 2 >= 0 && y + 2 < rowCount.length) {

// Only add nodes that found a depth within the bounds of the

// grid

GridLocation gridLocation = **new** GridLocation(rowCount[y + 2]++,

y);

gridLocations.put(node, gridLocation);

}

}

**return** gridLocations;

The basic concept of this algorithm is:

1. Go through all of the node links, picking out any links that contain the node that was selected
2. Add the nodes that are on the other side of the found links to a data structure
3. Find all of the nodes attached to the other side of links that are attached to the nodes found above (we will call the nodes above 1 level out, and these nodes 2 levels out)
4. Go through all of the found nodes and determine the minimum number of links it takes to get from the selected node to that node
   1. This is determined in a directional manner; that is, if a link goes from one node to another, that is a +1 traversal, if it goes the other way then it is a -1 traversal
   2. Example 1: The selected node links TO node A, which then links TO node B; this is a depth of 2 to node B, and a depth of 1 to node A
   3. Example 2: The selected node links TO node A, and link B links TO node A; this is a depth of 0 to node B, and a depth of 1 to node A
   4. Example 3: Node A links TO link B and node B links to the selected Node; this is a depth of -2 to node A, and a depth of -1 to node B
5. The nodes are added to the grid in 5 different rows, with each row signifying a depth from the selected node
   1. From the top row to the bottom row, the rows have depths +2, +1, 0, -1, -2, and the row with depth 0 will also contain the selected node
   2. In the order that they are parsed, each row is filled starting with the middle (except for the row of depth 0 that has the selected node at the middle location), and the continues to fill the row by going to the right of the middle, then to the left, and right, and left, etc.

## 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. This, actual is more than necessary because the draw method is called very frequently, but the realignment only needs to happen when a node changes “views,” which usually just means the node is disappearing/reappearing or that the node’s description is being shown/hidden (this is done using Node.showView()). Unfortunately, even when the realign method is only called when this happens, there are still glitches, so this should be looked into further to try to increase performance. Basically, this realignNodeText() method just finds out how to adjust the size of the rectangle, as well as if the description and/or title should be shown, as well as where they should be placed. Most of this is done relative to the rectangle to make movement simpler. It is important to note that ZVTM has two main methods for moving glyphs, Glyph.move(int x, int y), and Glyph.moveTo(int x, int y), which move the glyph relative to its current location, and move the glyph to an absolute and specific location, respectively.

## 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 rows in a word is determined by the static variable WORDSPERROW. This method is also where the textContainerHeight and textContainerWidth are determined based on the longest row and the number of rows.

**private** **void** drawTheString(Graphics2D g) {

String[] words = text.split(" ");

textContainerWidth = 0;

FontMetrics fm = g.getFontMetrics();

**if** (words.length > *WORDSPERROW*) {

**int** fontHeight = fm.getHeight();

**int** curWordPosition = 0, rows = 0, highestCharCount = 0;

String lineToWrite;

**while** (curWordPosition < words.length) {

**if** (curWordPosition + *WORDSPERROW* < words.length) {

lineToWrite = buildString(words, curWordPosition, curWordPosition + *WORDSPERROW*);

g.drawString(lineToWrite, 0f, rows \* fontHeight);

} **else** {

lineToWrite = buildString(words, curWordPosition, words.length);

g.drawString(lineToWrite, 0f, rows \* fontHeight);

}

**if** (lineToWrite.length() > highestCharCount) {

textContainerWidth = (**long**)fm.getStringBounds(lineToWrite, g).getWidth();

highestCharCount = lineToWrite.length();

}

curWordPosition += *WORDSPERROW*;

rows++;

}

textContainerHeight = fm.getHeight() \* rows;

} **else** {

textContainerWidth = (**long**)fm.getStringBounds(text, g).getWidth();

textContainerHeight = (**long**)fm.getStringBounds(text, g).getHeight();

g.drawString(text, 0.0f, 0.0f);

}

}

**private** String buildString(String[] strArray, **int** start, **int** end) {

String str = "";

**for** (**int** i = start; i < end; i++) {

str += strArray[i] + " ";

}

**return** str;

}

drawTheString() works as follows:

1. If the number of words is greater than the max words per row (WORDSPERROW), then multiple rows need to be drawn
   1. While the curWordPosition is an index < then the length of the text
      1. If the curWordPosition plus the next number of words for a row is less than the length of the text, then draw the next WORDSPERROW
      2. Otherwise draw the remaining text
      3. If the width of the row just drawn is the largest width, set the textContainerWidth to that value. This size is determined by the actual size of the characters that are drawn
   2. After the loop, the textContainerHeight is determined by the height of the characters multiplied by the number of rows
2. Otherwise just draw the string

## Navigating to Web Pages (zoom level-3, 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.

# Uploading Applet

When putting the applet on a server, it is important to change controller.Configuration.getNodesFilePath() to reflect the new location of the content.xml document.

# 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