RICHMOND School of Arts & Sciences

University of Richmond
UR Scholarship Repository

Math and Computer Science Technical Report Series

Math and Computer Science

3-2003

Creating and Modifying Dynamic Animation Sequences Using the TGT_Toolkit

Ross Gore

Follow this and additional works at: http://scholarship.richmond.edu/mathcs-reports

 Part of the Computer Sciences Commons

Recommended Citation

Ross Gore. *Creating and Modifying Dynamic Animation Sequences Using the TGT_Toolkit*. Technical paper (TR-03-01). *Math and Computer Science Technical Report Series*. Richmond, Virginia: Department of Mathematics and Computer Science, University of Richmond, March, 2003.

This Technical Report is brought to you for free and open access by the Math and Computer Science at UR Scholarship Repository. It has been accepted for inclusion in Math and Computer Science Technical Report Series by an authorized administrator of UR Scholarship Repository. For more information, please contact scholarshiprepository@richmond.edu.

Creating and Modifying Dynamic Animation Sequences Using the TGT_Toolkit

University of Richmond Math and Computer Science
Technical Report TR-03-01
Ross Gore
March, 2003

Introduction

Creating and modifying the dynamic animation sequences within the TGT_Toolkit is not for the faint of heart. This report contains advice, guidelines and refers to an example of how I went about constructing the dynamic animation sequences within the Linked Lists Tutorial. I have identified three major areas of that seem to be common to all dynamic animation sequences, but distinct from simply extending TGT_Slide as most other slides do; these areas are: dynamically creating animation configurations, providing "undo" and slide saving functionality and sequencing saved slides once they are properly configured. In these areas "slide saving" and "saved slides" individual steps that make up an animation sequence as opposed to a TGT_Slide instance. The TGT_AnimatorBox is not necessarily a dynamic animation tool. In normal use, TGT_Animator Box follows a fixed sequence of instructions from a configuration file that moves the images around the screen. We chose to extend this functionality because we felt static animation significantly limited user interaction.

Note: The following explanation assumes that you are not familiar with the Digital Tapestry Tutorial on linked lists. If you are you can skip this description of the slides containing dynamic animation:

Description of Linked List Tutorial Workshops

Adding new values is one of the basic tasks that all data structures support. The List class presented to students in the tutorial gives us two fairly straightforward ways to add a new value to our list: we can either insert at the front of the list or at the rear of the list. The developers have created two interactive "workshops" to allow students to try to insert a new value at the beginning of the list and at the end of the list. There is a `Select Action' list that contains a number of typical actions that students would use in developing methods that manipulate linked structures; such as creating a new node, changing the data value of a node, changing the ".next" field of a node, etc. The student's job is to build a sequence of these actions that will correctly insert a new value at the front of the list shown at the bottom of the window. As the user selects each action, he/she will be prompted for any further information needed, such as a name for the new node reference. The Java code corresponding to the selected action will be shown in a 'Source Code' area to the right when all of the necessary information for the actions has been collected. When students are satisfied with their source code, they can click the 'Execute' button to see the effect of their code on the example list. They can click on the Undo button at the bottom to remove the last action added to the 'Source Code' area. If they are stuck and want to see a correct solution, they can click on the 'Solution' button.

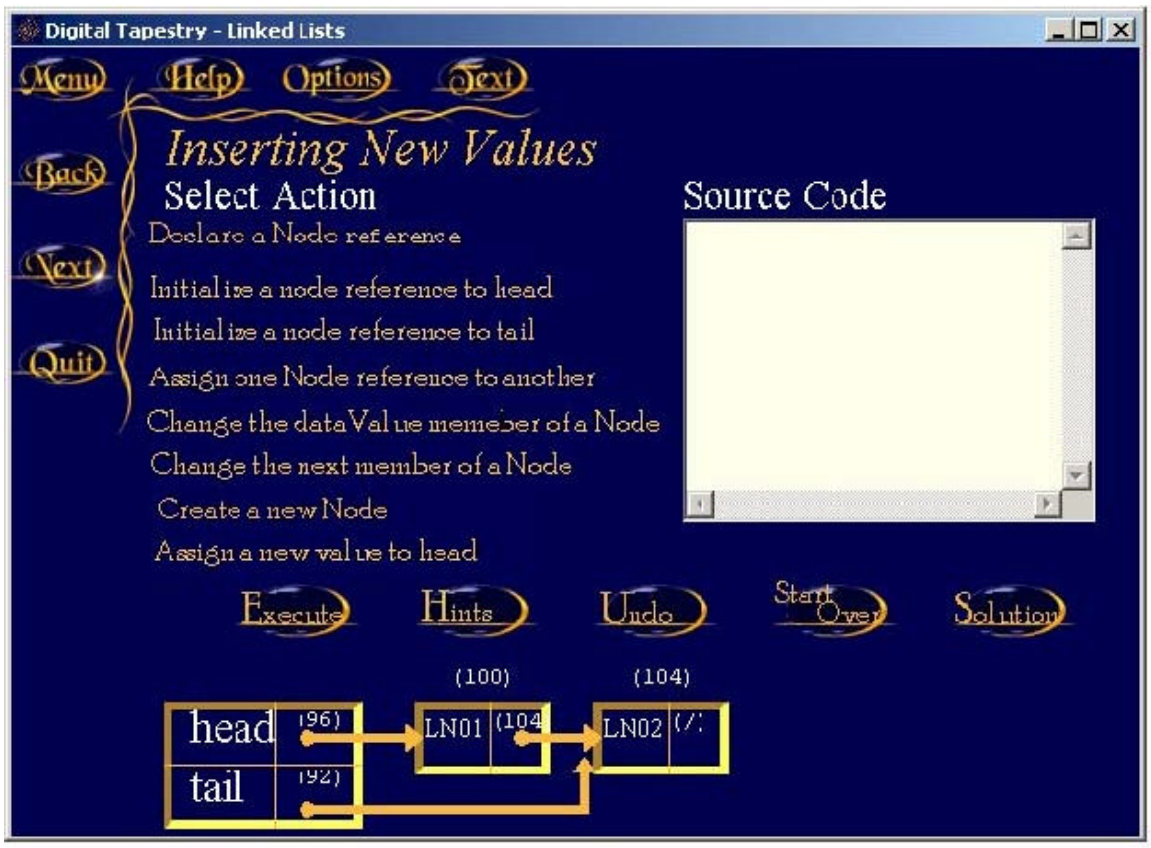

Figure 1: The Practice Workshop within the linked list tutorial.

Overview of Creating and Modifying Dynamic Animation Sequences

The details of this document can be overwhelming without an end in sight. Below is an outline of the procedure that this report documents. Being familiar with the structure of the report will help keep the reader from getting lost in some of the technical details.

Steps to creating dynamic animation:

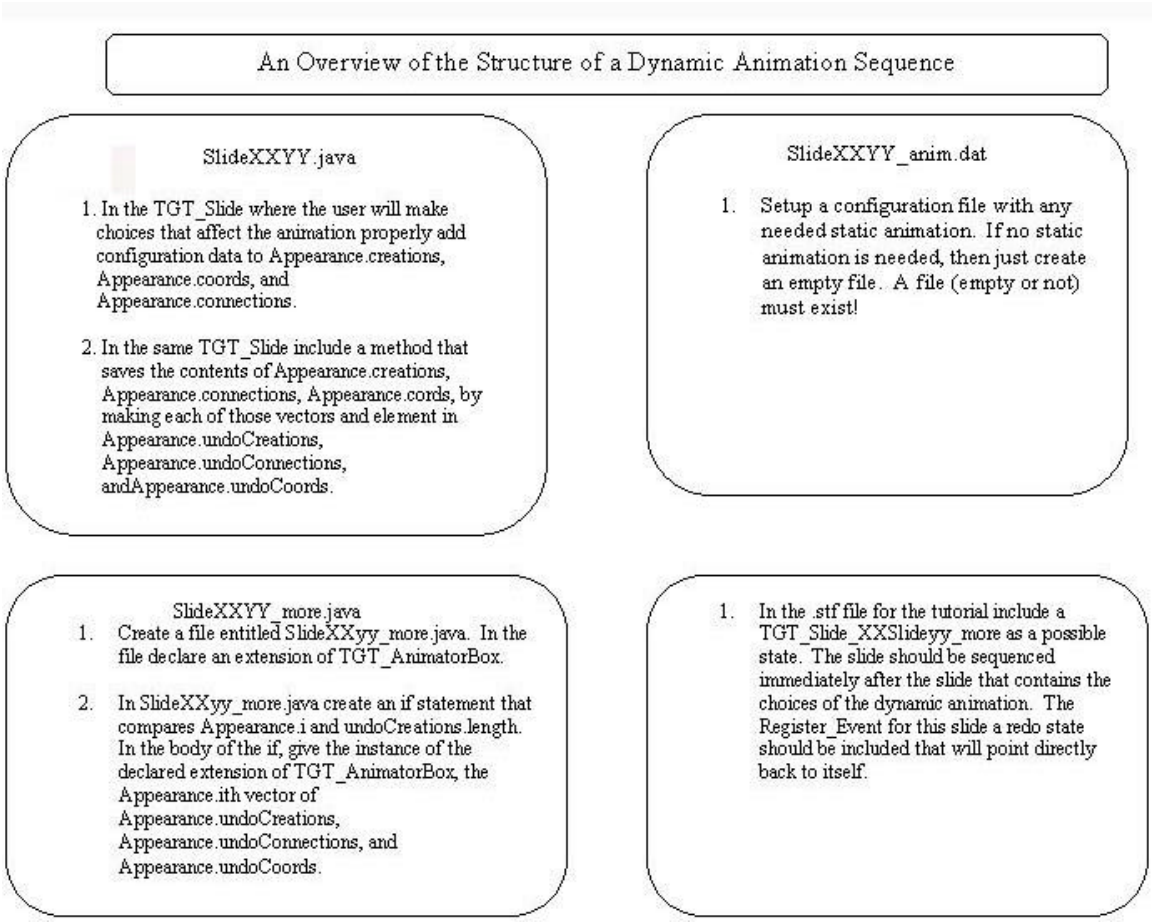

Figure 2: Overview of Dynamic Animation Structure

1. Set up a configuration file with any needed static animation. If no static animation is needed, then just create an empty file. A file (empty or not) must exist!
2. In the TGT_Slide where the user will make choices that affect the animation, add the appropriate configuration data to Appearance.creations, Appearance.coords, and Appearance.connections vectors.
3. In the same TGT_Slide include a method that saves the contents of Appearance.creations, Appearance.connections, Appearance.cords, by making

each of those vectors elements in Appearance.undoCreations, Appearance.undoConnections, and Appearance.undoCoords.

4. Create a file entitled SlideXXyy_more.java. In the file declare an instance of a specific extension of TGT_AnimatorBox for the new "more" slide.
5. In SlideXXyy_more.java create an if statement that compares Appearance.i and undoCreations.length. In the body of the if, give the instance of the declared extension of TGT_AnimatorBox, the Appearance.ith vector of Appearance.undoCreations, Appearance.undoConnections, and Appearance.undoCoords. This if statement will work like a loop since it will be executed for each step of the users code. The vectors are passed to the specific extension of the TGT_AnimatorBox in the XXSlideyy_more.java slide.
6. In the .stf file for the tutorial include a TGT_Slide_XXSlideyy_more as a possible slide. The slide should be sequenced immediately after the slide that contains the choices of the dynamic animation. The Register_Event for the XXSlideyy.java is a redo state will point directly back to itself.
7. In SlideXXyy_more.java add a setStateManager method to override the usual manager given to a TGT_Slide. The method will allow the slide to have the redo functionality that is in the .stf file. The code for the method will be almost identical to:

```
// Since we are using a redo we have to override the
// stateManager for this particular slide
public void setStateManager(TGT_StateManager sm) {
      // Placement of our "Step" button with redo functionality
      imgCI.addMouseListener(new TGT_StateListener("redo", sm));
```

```
        super.setStateManager(sm);
        // Everytime we click “Step” we get a
        //new snapshot of the code.
        Appearance.i++;
    }
```

These snip bits of code are part of an example tutorial called Test Sprites. Test Sprites can be downloaded from http://www.mathcs.richmond.edu/~lbarnett/TGT/index.php.

Dynamically Creating Configuration Data

When being used to create predefined animation the TGT_AnimatorBox reads in information from a configuration file that specifies what sprites will exist and how they will move. When being used dynamically the TGT_AnimatorBox begins the same as it does in the predefined case by reading in the same information from a configuration file. However, the TGT_AnimatorBox has the additional capability to add configuration statements when a slide is being displayed at run-time by making additions to the information stored in three vectors in Appearance.java. These vectors are: Appearance.creations, Appearance.coords, and Appearance.connections. Appearance.creations corresponds to the name, type, height, width, primary and secondary colors, and in and out connection points of a sprite. Appearance.coords contains the placement and movement information for the sprites. Appearance.connections corresponds to how the sprites “link up” to one another. For example an entry in this vector might be “LN01 `hasOutgoingConnection 0 to LN02 0 with SpriteDO1`”, where sprites LN01 and LN02 are TGT_ListNode Sprites and SpriteDO1 is a TGT_Connector Sprite. Within the file Appearance.java they are declared as:

```
public static Vector creations;
public static Vector coords;
public static Vector connections;
```

For more information on the exact parameters of all sprites that will be placed in these vectors please refer to the documentation of the TGT_Sprite class. The API documentation for the TGT_Sprite class is located on the Internet at http://www.mathcs.richmond.edu/~lbarnett/TGT/doc/. Here is an example of the simple methods within a TGT_Slide that creates sprite animation at runtime.

```
public SlideTS05() {
  // the name of the slide
  name = "SlideTS05";
  // the background image
  bg = "art/blue.gif";

  TGT_PointLayout pl = new TGT_PointLayout();
  this.setLayout(pl);
  // The sprite we are going to create dynamically
  Appearance.creations.addElement(
"SpriteIO3 TGT_ImageSprite art/goldcheck1.gif true 8");
// The rate, and movement of the sprite we just created
Appearance.coords.addElement(
"SpriteIO3 = 70,0:0,0:r5:0,0:70,70:r5:70,70:70,0:r5")
// There is only one sprite and it has no connections so
// Appearance.connections remains empty
// Create the extension of AnimatorBox with a configuration file
(ts_t/try_anim/TS05_start.dat) containing any
// static information, the Appearance vectors, the size of
// the size of the AnimatorBox and whether we want it to
```

```
// be written over.

anim = new TGT_AnimatorBox_TS05

("ts_t/try_anim/TS05_start.dat", myRootDir, Appearance.creations,

Appearance.coords, Appearance.connections, 400, 400, false);

// add the created AnimatorBox to the slide.

add(anim, new Point(0, 0));

// Set it to play.

anim.setPlay(true);

}
```

If a programmer wishes he or she can put all the configuration information in the file (TS05_start.dat), but the slide will not be able to interact with users. A programmer could also take the other route and store all the information in the vector that is read at runtime, however, it seems silly to put predefined information here, because the class would have to be recompiled for every change to the state animation configuration information. Below is the major body of the code for a specific extension of the TGT_AnimatorBox that is able to support both runtime sprite additions using both the TGT_Drawable sprites and previously created image sprites.

```
// read in the tokens from the vector and configuration //file
String temp = st.nextToken();
if (temp.equals("=")){
    String spriteImgFile = st.nextToken();
    if (myRootDir != null && myRootDir.length() != 0) {
        spriteImgFile = myRootDir + spriteImgFile;
    }
    if (TGT_Toolkit.behaviorFlags.get("DEBUG_IMAGELOAD")) {
        System.out.println("loading sprite image from " +
                           spriteImgFile);
    }
// if its an image sprite add to the list of images
// to be loaded into the media tracker the JMF uses
sprites.insertElementAt(new TGT_ImageSprite(spriteName,
TGT_Toolkit.getImage(spriteImgFile),
Toolkit.getDefaultToolkit().getImage(spriteImgFile), st.nextToken(),
Integer.parseInt(st.nextToken())), j);        }
// if its not an image sprite identify the class and
// instantiate the class
```

```
else {
      Class c = null;
      Constructor ctor = null;
   String spriteClassName = temp;
   if (DB_DAT) {
       System.out.println("spriteClassName = " +
                          spriteClassName)
   }
   TGT_AbstractSprite sprite = null;
// try to find class name
 try{
     c = Class.forName(spriteClassName);                              }
catch (ClassNotFoundException cnf){
        System.err.println("Couldn't find class " +
                           spriteClassName);
}
/ try to find class constructor
try{
    ctor = c.getDeclaredConstructor(new Class[]
    {String.class, StringTokenizer.class});
} catch (NoSuchMethodException nsm){
        System.err.println("Couldn't find constructor for"
                           +spriteClassName);
}
// create the new instance of the drawable sprite
try{
sprite = (TGT_AbstractSprite)
 ctor.newInstance(new Object[] {(Object) spriteName, (Object) st});
}
catch (InstantiationException e)
{       System.err.println("InstantiationException for " +
                           spriteClassName);
}
// catch all other errors our work may have caused
catch (IllegalArgumentException ire)
       {System.err.println("IllegalArguementException for "
                           + spriteClassName);}
catch (IllegalAccessException iae)
       {System.err.println("IllegalAccessException for " +
                           spriteClassName);}
catch (InvocationTargetException ite)
      {System.err.println("InvocationTargetException for "
                         + spriteClassName);
       ite.printStackTrace();
}
 // add drawable sprites to the sprite list
 sprites.insertElementAt(sprite, spriteCount+j);
}
// load all image sprites using the media tracker
if (sprites.elementAt(spriteCount+j)
   instanceof  TGT_ImageSprite){
   mt.addImage(((TGT_ImageSprite)sprites.elementAt
              (spriteCount+j)).getImage(), 0);
}
```

We now describe the use of the Appearance vectors with sprites from the TGT_Abstract_Sprite hierarchy. If you choose to develop using TGT_Image_Sprites, the following descriptions can be ignored. Apperance.coords corresponds to the exact placement and movement of each sprite within the animation sequence. It is particularly difficult to anticipate how the users actions should be moved on the screen. In the linked lists tutorial I only implemented movement in the standard solution that the slides offer. The final vector the TGT_AnimatorBox reads from is Appearance.connections. Here is the idea behind this configuration: after each code choice the user makes, the program stores the effects of that choice in the appropriate vector that the programmer has created in the program. Once the programmer has recorded the users choice in the new vectors, it be added to the Appearance vectors. If a user's choice creates a new sprite, an addition to Appearance.creations is made. If the user chooses to have the head pointer of the list point to a new node, head's connections are changed in Appearance.connections. For use in the list tutorials I used additional vectors to keep track of modifications made to the original animation and vectors to keep track of sprites the user would add. I abstracted these vectors out into the file ListModTools.java. The vectors origCreations, origConnections, and origCoords are used to record the original animation sequence that the slide begins with and the changes that may be made to them. The other vectors: createdConnections, createdNodeNames, createdNodeValues, createdNodeAddresses, createdNodeNextAddresses, and createdNodeInConns all are used to describe the user's previous choices.

With these tools in hand actually manipulating the animation to correspond to the user's code becomes more feasible. The simplest change occurs when no new sprites need

to be added into the sequence, a data value or some aspect of the appearance of the sprite may only need to be changed. Begin by reading in the sprite the user wants to change, then locating that sprite in the appropriate created or original sprites list and replacing the information with the appropriate changes. The job of the programmer becomes more difficult when adding sprites to the animation. Initially consider what the user wants done, and what features the sprite needs. Next, based on these features add the appropriate information to the corresponding createdNode Vectors. Finally comes the tricky part: "How is the rest of the list affected?" If a user wants a new node to point to the head, then an inConnection point and some connection configuration information needs to be added to the head to create and receive the connection. Begin by searching the origCreations vector for the head node; then replace the element you find with an element with one more incoming Connection Point. Now add the connection configuration information to the createdConnections vector. All of these steps must be done before the TGT_Animator_Box instance is created. Here is an example of the inserting a new node at the front of the list in the insertAtFront method in ListModTools.java. This is the correct solution to the Linked List workshop animation. Assume the head variables (headValue, headNode, headInConns, headOutConns, headAddr, and headNextAddr) have been globally declared and correctly maintained in ListModTools.java.

```
public static void insertAtFront(String right){

// gather the appropriate strings to use in our searches
// below

// just a string to make configuration statements either to code
String listWhc = "75 40 0xFABA40 white";

// Find the position of the node
```

```
int rightIndex = createdNodeNames.indexOf(right);

// Based on that index make all modifications
String rightInConns = (String)
createdNodeInConns.elementAt(rightIndex);

String rightOutConns = (String)
createdNodeOutConns.elementAt(rightIndex);

String rightValue = (String)
createdNodeValues.elementAt(rightIndex);

String rightAddr = (String)
createdNodeAddresses.elementAt(rightIndex);

String rightNextAddr = (String)
createdNodeNextAddresses.elementAt(rightIndex);

// find old head info and replace the connections to correspond to the
// newly inserted node

// find the index of the newly created sprite
int tI = Appearance.creations.indexOf(right +
" TGT_ListNodeSprite " + listWhc+ " " +rightInConns +" "+ rightOutConns
+" " +rightValue+ “ true " + rightAddr + " false " +"1 "+
rightNextAddr+ " false");

// set the element at the index we found above to the
// head of the list
Appearance.creations.setElementAt(right +
" TGT_ListNodeSprite " + listWhc+ " " + headInConns +" "+ rightOutConns
+" " +rightValue+ " true " + rightAddr + " false " +"1 "+
rightNextAddr+ " false", tI);

// We have correctly changed the Appearance.creations
// vector now we do the same thing for the connectors
tI = Appearance.creations.indexOf("SpriteDO2 TGT_RtAngleConnectorSprite
5 5 "+"0xFABA40 white true 4 RIGHT_DOWN");

Appearance.creations.setElementAt("
SpriteDO2 TGT_RtAngleConnectorSprite 5 5 "+
"0xFABA40 white true 4 UP_RIGHT", tI);


// We have finished updating our created sprite now we
// must update the previous head

// find the index of head
tI = Appearance.creations.indexOf(headNode+
" TGT_ListNodeSprite " + listWhc+ " " + headInConns +" "+ headOutConns
+" " +headValue+" true " + headAddr + " false " +"1 "+headNextAddr+ "
false");

// change head’s configuration information
Appearance.creations.setElementAt(headNode+ " TGT_ListNodeSprite " +
listWhc+ " " +"{ -5 20 10 -5 30 -5 25 50 }" +" "+ headOutConns +" "
```

```
+headValue+" true " + headAddr + " false " +"1 "+headNextAddr+ "
false", tI);

// Update the previous head’s connections
tI = Appearance.connections.indexOf("SpriteHN01 "+connectStr+" 0 to
"+headNode+" 1 with SpriteDO2");

Appearance.connections.setElementAt("SpriteHN01 "+connectStr+" 0 to
"+right+" 0 with SpriteDO2", tI);
}
```

Undo and Slide Saving Functionality

Since we want the student to be able to step forward and back we still need to store the state our of slide so we can show the user what happened during the execution of this particular code choice, and we need to be able to return to the previous states later if the user chooses to undo several future steps. After each step of the animation, which corresponds to a snapshot, is completed the programmer should assemble the appropriate configuration from the original and created Sprite vectors and store the information in Appearance.creations, Appearance.coords, and Appearance.conns. Once the appropriate configuration to recreate a specific snapshot of the user's code exists, we just have to save it. The Apperance.creations, Appearance.coords, Appearance.conns vectors should be placed in the separate vectors in Appearance.java :Apperance.undoCreations, Appearance.undoCoords, and Appearance.undoConns. The ith element of these vectors will contain the ith state of the student's code, or ith snapshot of the data structure the user is programming. These stacks and the index "i" are declared in Appearance.java as:

```
public static Vector undoCreationsStack;

public static Vector undoCoordsStack;

public static Vector undoConnectionsStack;

public static int i;
```

Adding the individual elements into these vectors from the specified vectors is done by:

```
undoCreations[Appearance.i]    =  Appearance.creations;,
undoCoords[Appearance.i]       = Appearance.coords;
undoConnections[Appearance.i]  = Appearance.connections;
Appearance.i++;
```

When the user chooses to "undo" a step, the programmer "pops" the top element off each of these vectors.

Sequencing Slides once they are Saved and Properly Configured

Once we have captured all the different "snapshots" of student's code we need to sequence the snapshots so they can be displayed. In the .stf file for the tutorial unit, a more slide needs to added to go directly after the slide that provides an interface for code entry.

Here is an example from the Linked Lists tutorial .stf file:

```
SlideLL09 {
        Register_Event  prev    SlideLL08
        Register_Event  next    SlideLL10
        Register_Event  menu    SlideLL02
        Register_Event  more    SlideLL09_more
        Register_Event  redo    SlideLL09
        Register_Event  optn    TGT_OptionSlide
}
SlideLL09_more {
        Register_Event  prev    SlideLL09
        Register_Event  next    SlideLL10
        Register_Event  menu    SlideLL02
        Register_Event  redo    SlideLL09_more
        Register_Event  optn    TGT_OptionSlide
```

```
}
```

A “more” slide allows the programmer to present additional information related to a slide in within a topic. In the case of an algorithm workshop we use SlideLL09 to display the code the user has input, and SlideLL09_more to show the snapshots of the users code at each step of execution. After each snapshot the user will want to step to the next snapshot of code. This is the "redo" state in the shown above .stf file. “Redo” is a tag that corresponds to a specific state transition in both the TGT_StateManager and the .stf file that serves as a finite state machine for the linked lists tutorial. The redo state goes to itself but each time loads the next set of vectors into the TGT_AnimatorBox to create a different snapshot. The implementation of this "more" slide requires extending the TGT_AnimatorBox to a TGT_AnimatorBox specific to the slide. This is very similar to the way a slide within a tutorial extends TGT_Slide:

```
TGT_AnimatorBox_LL09 extends TGT_AnimatorBox();
```

The extended animator box has the capability of adding configuration statements to create unique animation sequences at runtime. This specific animator box takes three vectors as parameters; these three vectors are vectors of vectors should be the ith vectors within Appearance.undoCreations, Appearance.undoConns, and Appearance.undoCoords. Each displayed slide will have a step button, that will increase the snapshot by one by invoking the "redo" state. Every time the button is clicked the programmer only needs to increase the global index Apperance.i and then load a specific new TGT_Animator_Box instance with the indexed elements in the undoVectors. Here is an example:

```
// the loop part of this "if" statement comes because this
```

```
// executed everytime the user chooses to "step" to the
// next snapshot
if (Appearance.i <maxDisplayState){
 anim = new TGT_AnimatorBox_LL09
("ll_t/slide_anim/LL09_start.dat", myRootDir,
undoCreations[Appearance.i], undoCoords[Appearance.i],
undoConnections[Appearance.i], 600, 330, false);
add(anim, new Point(0, 0));
anim.setPlay(true);
}
// Since we are using a redo we have to override the
// stateManager for this particular slide
public void setStateManager(TGT_StateManager sm) {
// Placement of our “Step” button with redo functionality
imgCI.addMouseListener(new TGT_StateListener("redo", sm));
super.setStateManager(sm);
// where the loop gets incremented, so now everytime we
// click “Step” we get a new snapshot of the code.
Appearance.i++;
}
```

Conclusion

The most difficult part of creating dynamic animation sequence is creating proper configuration vectors to the user specifications. Limiting the number of times the user can change original values, and thoroughly error checking all requests to make sure every action involves a sprite that actually exists can make the programmer's job much easier. Despite these warning by keeping proper organization the programmer can achieve impressive dynamic results that significantly improve user interaction.