Design and Implementation of Interactive Tutorials for Data Structures[1]

Ross Gore and Lewis Barnett

Abstract— The Tutorial Generation Toolkit (TGT) is a set of Java classes that supports authoring of interactive tutorial applications. This paper describes extensions to the capabilities of the TGT and several new tutorials aimed at the Data Structures course which were built using the toolkit.

I. Introduction

The Tutorial Generation Toolkit is a set of Java classes that implements a framework for developing interactive tutorial applications in Java. The basic form of a tutorial is an interactive slide show combining explanatory material and exercises that provide students with immediate feedback. The toolkit provides facilities for flexible sequencing of "slides," use of Java GUI components to compose slide appearance, simple animation of fixed images, voice-overs, and multiple choice self-tests with results logged to a database if desired. A tutorial author creates a tutorial by writing a set of Java classes that subclass the TGT_Slide class, which is itself a subclass of java.awt.Panel. Each of these classes can use any of the Java GUI components to present information or solicit user interaction. The original distribution contained a fairly complete set of tutorials on CS 1 topics. The TGT is described more completely in [1].

The work described in this paper represents a progress report on an effort to develop tutorial materials to support teaching of the Data Structures course. It was clear that the capabilities of the existing toolkit, particularly the support for only simple image-based animation, would not be sufficient to support the new tutorials. The focus on data structures thus led to the following design goals:

- the tutorials should utilize and extend the already established TGT framework;
- the TGT framework should be modified to make it easier to add new tutorials; and
- the new tutorials should provide graphic visualizations based on user input.

In trying to meet these goals we decided to break away from the original packaging strategy for the tutorials. After creating a more flexible way to add new tutorials, we designed and implemented four tutorials that present topics to users in an interactive and visual environment. While the TGT framework had already established a basis for the creations of the tutorials, the more advanced topics we were trying to present required both extending and modifying parts of the framework. These changes created a more flexible environment that allowed us to give users more control over actions within the tutorials. This paper describes the design decisions that governed the extensions and modifications of the framework, as well as design of the tutorials themselves. The overall design of our project fell into three categories: reorganizing the packaging of tutorials, extending the capabilities of the toolkit and determining the content and organization of individual tutorial units. The next three sections of the paper address these areas. Information about obtaining the tutorials and the TGT package is included in the last section.

II. Modification of Tutorial Packaging

The original distribution consisted of a set of tutorials for CS 1 bundled as a single application with a beautiful "menu screen" that allowed the user to select which of the tutorials they wanted to work through. The resulting application thus consisted of the classes that implemented all of the individual tutorials and the shared toolkit

[1] This work was supported by a Collaborative Research Grant from the University of Richmond and by NSF Grant DUE-9652982.

classes, plus additional data files (for voice-overs, etc).

This presented two significant problems. First, it did not allow for any additional units to be added to the bundle without the significant overhead of redesigning the menu screen, which was essentially an image map with interactive highlighting. This was inconvenient, and the problem would only become larger and more time consuming with every additional unit. Second, the packaging did not have the flexibility of allowing tutorial units to be distributed individually. If a user were only interested in one tutorial unit, they would still have to download the entire bundle. Again, this was undesirable, as the size of the bundle would only grow with continued additions. To solve both of these problems, we decided to "break out" each tutorial from the bundle, into its own separate package. While this required that each tutorial have its own copies of the files shared by all of the tutorials, it made the addition of tutorial units and individual tutorial unit distribution extremely simple.

III. TGT Framework Modifications and Extensions

The major change we decided to make was in the design of the animation system structure of the class TGT_AnimatorBox. This class originally provided a framework for a user to create animation sequences from previously created images, referred to as "sprites." However, this approach came with quite a few serious limitations. The type of animation supported was that of moving previously created images around the screen according to commands in a configuration file for the TGT_AnimatorBox instance. This approach works reasonably well for planned animation sequences such as algorithm animation for fixed inputs, but was simply not capable of the kind of animation controlled by user interaction that we had in mind for the new tutorial units. The focus of our enhancements to the animation facility fell into three areas. First, we needed sprites that could draw themselves rather than relying upon an image file for their appearance, thus allowing the appearance of a sprite to change over the course of an animation if necessary. Second, we needed to be able to change the set of sprites that an animation was using on the fly. Finally, we needed to be able to adjust the movements of the sprites as the animation progressed in response to user input, rather than simply following a fixed script read from a configuration file.

A. Self Drawable Sprites

The typical way of illustrating many data structures is the "box and arrow diagram," where nodes in the structure are indicated by a box, and links between nodes are shown as arrows. Consider the task of deleting a node from a linked list from the point of view of animating the steps required. We start with three nodes and some connections among them. During the course of the deletions, the links from the two adjacent nodes may be changed, and one of the nodes is "moved out" of the list. In order to support this type of animation without having to produce a huge number of image files for nodes with different data member values and pointers of various lengths and in various orientations, we needed for our sprites to support the notion of connections that could automatically redraw themselves when the sprites they were connected to moved. We also needed to be able to change which sprites connections were attached to during the animation, and to change the "contents" (i.e. the values stored in a node, or the addresses corresponding to links, etc.) of the nodes on the fly.

We began by redesigning the TGT_Sprite class to fit into an inheritance hierarchy of sprite types which would include a new type of sprite called a "drawable sprite." These new sprites would create their own appearance using the drawing primitives from the Java Graphics classes rather than displaying an existing image. Special purpose classes were derived from the basic drawable sprite class for linked list nodes and "connectors" (used for pointers between nodes), along with a helper class that acted as an attachment point in the nodes for the pointers. Drawable sprites can still be moved around, and when they are, any connectors attached to them redraw themselves in specific ways. We did not include a general edge routing algorithm for redrawing connections between sprites, but instead subclassed the connector sprite for straight connections, connections with right angles, and so forth. This approach was less complex and in addition gives the slide programmer an added measure of control over the appearance of the animation.

With this arrangement, we were able to proceed with development of tutorials involving linked list data structures, and we had also left the door

open for the development of other node-like sprites such as binary tree nodes. Figures 1 and 2 show the organization and use of some of the new sprite classes. The current sprite hierarchy is shown in Figure 1. All of the classes implement the TGT_AbstractSprite interface, which dictates a draw method as well as accessors and mutators for properties that all implementing sprites are expected to provide. The TGT_Sprite class is now an abstract class that contains definitions of many of the common methods, such as accessors and mutators for the properties that all sprites must support. The draw method remains abstract in this class. The old TGT_Sprite class which supports animation of images from files is now called TGT_ImageSprite.

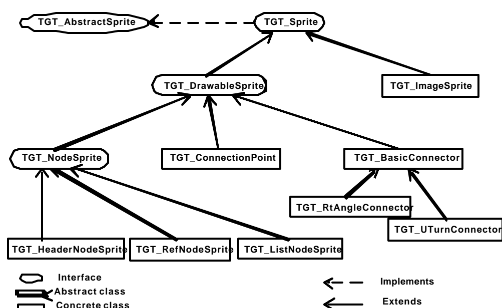

Figure 1: TGT_Sprite Hierarchy.

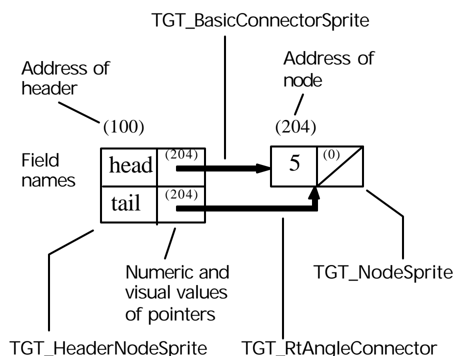

Figure 2: Sprites for constructing linked lists.

An example of special-purpose sprites for displaying linked lists is shown in Figure 2. To facilitate making the cognitive connection between the diagram and the allocation of dynamic structures in main memory, the node sprites can display their pointer values both visually as arrows from the source of the pointer to the referenced object and as the memory address (shown in parentheses) actually stored in the "next" or "head" field of a node. Display of the addresses can be turned off. The TGT_HeaderNodeSprite represents the head of a linked list maintaining head and tail pointers to nodes within the list. The TGT_NodeSprite has a data field and a "next" field, which is null in the example shown here. Two of the "connector" sprites are shown, a straight arrow and an arrow with a right-angle bend, which can be oriented in any of the eight possible directions (think of the knight's moves in chess). Not shown are the TGT_ConnectionPoint instances contained within each of the node classes. These objects serve as anchors for the connectors. The connection points have positions relative to the position of the sprite they belong to, and connectors anchored to them take their endpoint positions from the connection point objects. When a sprite with connection points is moved, the connectors attached to the connection points are automatically redrawn to reflect the movement.

B. Run-time Sprite Additions

We felt one to the most important aspects our tutorial on data structures should provide students is the ability to visually represent the actions of their structures. We decided we wanted to create a "workshop" where students could perform different actions, for example, operations on a linked list, and then see the individual results of each of their actions, step-by-step. What we envisioned was a way to allow students to piece together code snippets into an algorithm, and to then animate the outcome of their work, whether correct or not. Originally, TGT_AnimatorBox did not provide any capabilities to modify an animation sequence once the configuration file for that sequence had been read. We wanted to be able to create an environment where we could display unique animation sequences based on user input. To do this using the old framework, we would have had to create an animation configuration file for every possible combination of user choices, and then pick the correct one to display. We wanted to allow users many different choices, the results of which often depended upon the previous choices. For any reasonably complex task, such as adding a node to a linked list, the number of possible combinations, and thus the number of "canned" animation sequences required, is unworkable. To avoid this problem, we

subclassed the TGT_AnimatorBox to support runtime additions of sprites based upon user input. This allowed us to create an environment for individual TGT_Slides where we could add or remove sprites as the user made their choices. The subclass can accept animation data in the form of vectors of sprites rather than as descriptive information from an existing file. When the user was done, we could create the vectors to for the animation from their choices.

C. *Dynamic Animation Sequences*

While the previous extension allowed to us to create a unique animation sequence on the fly based upon user input, once the sequence was created it was still static. The remaining requirement was the ability to let a student single-step through the sequence of actions they had constructed, animating each step based on the state of the previous step. The dynamic animations in the Linked List tutorial described below also take advantage of the fact that the TGT_Slide class supports the notion of "sub-slides," which are basically a set of panels that can be successively displayed within a single TGT_Slide instance. So, each step through a student's constructed algorithm involved creating the vectors for an animation that was displayed on a subslide. Even with these enhancements, the kind of interaction present in these tutorials requires painstaking work to set up, and once an animation is in progress, there is no real way to change its course. What we have done is provided a way for an application to construct and play a sequence of animations on the fly.

IV. NEW TUTORIALS

The Data Structures course is typically where students are introduced to dynamically allocated linked data structures and some of the more sophisticated and efficient sorting algorithms. The current set of tutorials addresses these two topics. We chose to break up the sorting subject matter into three tutorials on different sorts.

A. *Insertion Sort*

We decided it was necessary to cover one basic and fairly straightforward sorting algorithm. This tutorial would not only cover the details of the algorithm itself, but introduce students to terminology and common themes shared by sorting algorithms. This tutorial could be used as background in a Data Structures course or to introduce this algorithm in a CS 1 course.

B. *Quicksort*

We felt it was necessary to cover Quicksort, because it is regarded as the one of the fastest and most useful sorting algorithms. The tutorial first traces through the top-level code for Quicksort while showing its effects on a small array, without going into the details of the partitioning algorithm. The student can either single-step through the code or let it run at a fixed pace. Next, the partitioning algorithm is demonstrated by providing an array and allowing the student to interactively pick the pivot index. The partition algorithm with the student's selected index is then simulated to show how balanced the subarrays would be using the student's selected index. Finally, we wanted to show students various Quicksort optimizations.

C. *Heap sort*

The last sort we chose to present was heap sort. While heap sort is usually not thought of as the fastest sorting algorithm, it is still very efficient, and presented us with a great opportunity to display animation and graphics to accompany the source code. Without illustrations to visually depict what the algorithm is doing, it is very difficult to fully understand this sort. Also, heap sort gave us an opportunity to familiarize students with the "heap" data structure and introduce them to trees. We felt this would be a good stepping stone to build on for some of our future data structures tutorials.

D. *Linked Lists*

The new tutorial on linked lists was the generator for most of the new development in the tutorial framework described earlier. Programming linked structures for the first time is often difficult for students, and much of the difficulty arises from confusion about how pointers (or references, depending on what your language chooses to call the construct) work, what using them in various ways means, and what the effects of code that modifies pointers look like.

We wanted to address these problems by providing an interactive workbench for playing with code that operates on linked lists. By allowing the student to "write the code" to add or delete nodes and traverse lists, and then animating their code for them, students could "see" the results of common mistakes and

develop a deeper understanding of programming dynamic structures. The tutorial first asks students to choose the condition for a while loop that traverses an existing linked list from a set of candidate conditions. The candidates include conditions that result in “off-by-one” errors in both direction, causing both references through null pointers and missing the last element of the list. Subsequent exercises give students the opportunity to build their own code fragments to for the “insert at front” operation. The code is constructed interactively from operations like “create a new node reference,” “set the value of the next field,” “change the value of the list’s head reference,” and so forth, with dialog boxes used to collect user input such as the name of the node to modify or create. The student is then allowed to step through the code on an example list to see whether their code correctly performs the insertion. It is actually incorrect solutions that are most effectively demonstrated through this exercise, as incorrectly initialized reference variables, self-loops, and omitted operations like resetting head or tail are clearly demonstrated. A correct solution is available in case of mounting frustration.

V. Future Work

While the packaging modifications simplified extension and distribution of the tutorials, each tutorial still consists of a Java application and a large number of ancillary files including image files and sound files. The ultimate goal is to come up with a good Web-based distribution system that will circumvent any installation issues that users may experience. Java Web Start is an obvious candidate, but there are difficulties. In order to use Java Web Start, an application must be completely packaged in a JAR file, and at present, it is not possible to play sounds that are stored in a JAR file directly.

We are also interested in developing tutorials on other data structures, as well as tutorials for topics from architecture and algorithms.

VI. Conclusions

Creating interactive tutorials with the toolkit described in this paper is not for the faint of heart; commercial packages such as Macromedia Director allow many of the same effects. What the TGT classes provide are a kind of complete flexibility. If you can program your idea in Java, the TGT classes provide a framework in which to implement your idea. This paper documents significant improvements in the packaging strategy used by the toolkit and increased flexibility and power in the TGT_AnimatorBox and the corresponding TGT_Sprites it supports. These improvements in the animation and display capabilities leave us with a tool that will allow development of further highly interactive tutorials in the area of data structures and algorithms.

The modifications to the tutorial framework can also be construed as an object lesson for software designers. All of the modifications required for the current set of tutorials sprang from a failure to pay appropriate attention to potential changes in requirements over the life of the software system during the original design phase. A packaging strategy that was driven more by esthetic concerns than by thoughts of potential extensibility had to be replaced. An animation facility with a configuration policy that proved to be too restrictive was expanded to allow a kind of dynamic creation of animation instances. In terms of the sprites, the benefits of using interfaces and abstract classes to separate the construction of special purpose sprites from the mechanism of displaying sprites is evident, and has left us with a greatly improved framework for the development of future sprite subtypes.

VII. Getting the TGT Package and Tutorial Units

Information about downloading the TGT package with documentation is available by following the TGT links from (author’s web site).

References

[1] L. Barnett, J. Casp, D. Green and J. Kent. Design and Implementation of an Interactive Tutorial Framework. *Proceedings of the 29th SIGCSE Technical Symposium*, Atlanta, Georgia, February 25 – March 1, 1998. Pages 87 – 91.