YAAS: A FRAMEWORK FOR AUTOMATIC ANIMATION

By

William Hipschman

Honors Essay

Computer Science

University of North Carolina

4/9/2012

Approved:

____

Abstract

Historically the automatic generation of algorithm animations is difficult. In this paper we describe Yet Another Animation System (YAAS), a framework that automatically generates list-based algorithm animations and that can be generalized to automatically produce animations for any observable data structure. By defining key methods of observable data structures as interesting events, we can respond to any change in the data structure by handling fired events, and so create animations for any algorithm comprised of such operations. Pairing the handling of these events with graphical manipulation, we are able to create useful animations based solely off of algorithm code. Our system improves programmer tools by providing fully-reversible animation, user-defined data support, and high levels of customizability, while maintaining strict separation of concern between components.

1. Introduction

All aspiring computer scientists must at one point during their academic career learn about data structures. Typically this is done using a textbook that runs through the most common structures: lists, stacks, queues, graphs, hash tables, etc. Often this course includes, or is required alongside a course that includes, the algorithms involving these data structures.

These courses employ textbooks that describe these algorithms and structures in words and pictures. Although these pictures are quite accurate, they are static. If a student doesn’t understand the concept, looking at the picture may not help. They are not interactive.

Ideally a student would see not merely a generic snapshot of a data structure and its algorithm, but a view that changes according to the user's needs. Many software packages have been developed with this goal in mind. They provide a wide range of visualization options and algorithm support.

There are many aspects of a program that could be animated. The literal code could be animated [16, 21, 42], or the data structure [9, 10, 40], or an algorithm could be viewed [5, 27, 30, 31, 37, 38]. We provide support for the animation of both algorithms and data structures. Henceforth we will refer to these together as 'algorithm animation.'

Algorithm animation is the process of abstracting a program's data, operations, or semantics, and creating dynamic graphical views of those abstractions [39]. Dynamic graphical views are graphical views that change over time; the graphical elements may move, adjust shape or size, be added or removed, or update any number of characteristics. Dynamic is an ambiguous word in computer science in that it can also refer to changes made at run-time. In this paper we will use 'run-time' or 'run-time animations' to indicate an animation that is being shown on screen at the same time that the relevant algorithm is being run. We reserve 'dynamic' to refer to on-screen graphical changes. An algorithm animation system, package, or toolkit is a piece of software that facilitates algorithm animation.

Algorithm animation systems are often used as educational tools, as presentation tools, as debuggers, or as GUI helpers [30]. Each of these applications has a different type of person as a user. We find it useful to make a distinction between the different types of users. The 'user' views the animation, the 'visualizer' is the system itself, the 'programmer' is a person who wants to create new animations using the system, and the ‘developer' is the designer of the system [30]. One person could, however, perform multiple roles; a programmer may also view the animation he/she creates, and so be a user. The developer may design new animations using the system and so also be a programmer.

The quality of an animation is often based on an empirical study of its effectiveness [13], but this is not the only measure of quality. We propose that the programmer’s time is valuable and so, in the same way that it is important to create a good visual representation for the user, it is vital to provide an easy development experience for the programmer. This means that an animation system must, in addition to providing informative animations, allow for animations that are easily created, extended, and modified. Assume a programmer is interested in creating animations of various sorting algorithms. The only code the programmer should have to write is the code for the algorithms themselves. If he/she decides to customize the graphics, the programmer should be able to make these changes in one place, without changing any part of the algorithms. If the user is viewing the animation of an algorithm and wants to modify it to see how the behavior changes, he/she should be able to do this easily, without restarting the animation.

2. Related Work

It is important to define the necessary components of an algorithmic visualization package in order to analyze them. Some common criteria include platform independence, easy usability, support for inclusion of source or pseudo code, support element highlighting and marking, user adjustable operation time, and wide applicability [16].

Platform independence allows widespread use of the package; since many different systems are used by students, educators, and presenters it is necessary that anyone should be able to run the software. Easy usability allows users to quickly become comfortable with the software, but is potentially troublesome to measure. Support for code and highlighting allows a cognitive mapping from algorithm behavior to code. This dual-encoding scheme, which represents an algorithm as both a visual process and as code, can be helpful for learning algorithms, but it is not as important as a system that allows for stepping backwards and forwards through an algorithm, or a system that supports cognitive constructivism [13]. User adjustable operation time is necessary to give the user high levels of control over the animation; the user should control when the next step of the animation takes place and how fast that step is animated. This is especially useful for algorithms that have many complex steps that need to be animated quite slowly. Wide applicability is perhaps the most obvious criterion; an algorithm animation package should have support for many different algorithms.

2.1 Additional Algorithm Animation Criteria

The above criteria assume that the primary person interacting with the systems is someone viewing animations, not someone creating his or her own animations; they seem to indicate that the animations are more often accessed by users than by programmers, or that the user views animations created for them as opposed to creating their own animations. It is entirely possible that the user is also the programmer.

These are good criteria for the actual animations; however, we find it useful to introduce several criteria for the animation system itself. These include reaction to runtime changes, the ability to construct new animations without understanding an algorithm, and, most importantly, separation of animation code from algorithm code.

Being able to react to run-time changes is important in most presentation settings, especially in classroom laboratory sessions. If a user views an animation that a programmer created and is presenting, it may be the case that they have a question about how the algorithm works. Assume that a list is being sorted using swaps and that all elements of the list have distinct keys. The user might be curious what happens if two keys have identical values. The person controlling the animation should be able to add an element with a duplicate key and resort the list. The controller should not have to restart the animation, recompile script or code, or create a new animation.

Algorithm animation systems are often used as teaching and debugging tools [13, 16, 42], and so it is to be expected that programmers may not perfectly understand the code they are visualizing. If, as is common [22, 40], the code controlling animation graphics is coupled with algorithm code then it is difficult for a programmer to create a correct animation when the algorithm code is flawed, and vice versa. It would be preferable if an incorrectly written algorithm was always animated such that the errors in the algorithm were evident graphically.

Separation of graphics and algorithm code is good practice. If separate components are coupled, then the code can only be used for animation; the algorithms can never be used in a different context. Further, it reinforces correct programming style which is especially important if the programmers are students. Finally, automatic animation is only possible if the programmer can write the code knowing nothing about graphics or animations. This necessitates the decoupling of graphics and algorithm code.

2.2 Current Practices

A system’s specification technique describes how the code is connected to the graphics. The most common techniques are event-driven, state driven, visual programming, and automatic animation [15]. An event-driven specification recognizes ‘interesting events’ in a piece of code and maps each event to a step in the animation. State driven methods divide the data the code executes on into separate states. Each of these states is mapped to a graphical frame. Visual programming animates the actual code itself; it includes data assignment, variable declarations, etc. It is different from other specification techniques in that it doesn’t focus on algorithms, and so it is not often used in algorithm animation systems. Automatic animation is the least common and most difficult specification technique to use. It automatically generates the animation from the given code with no extra work on the part of the programmer.

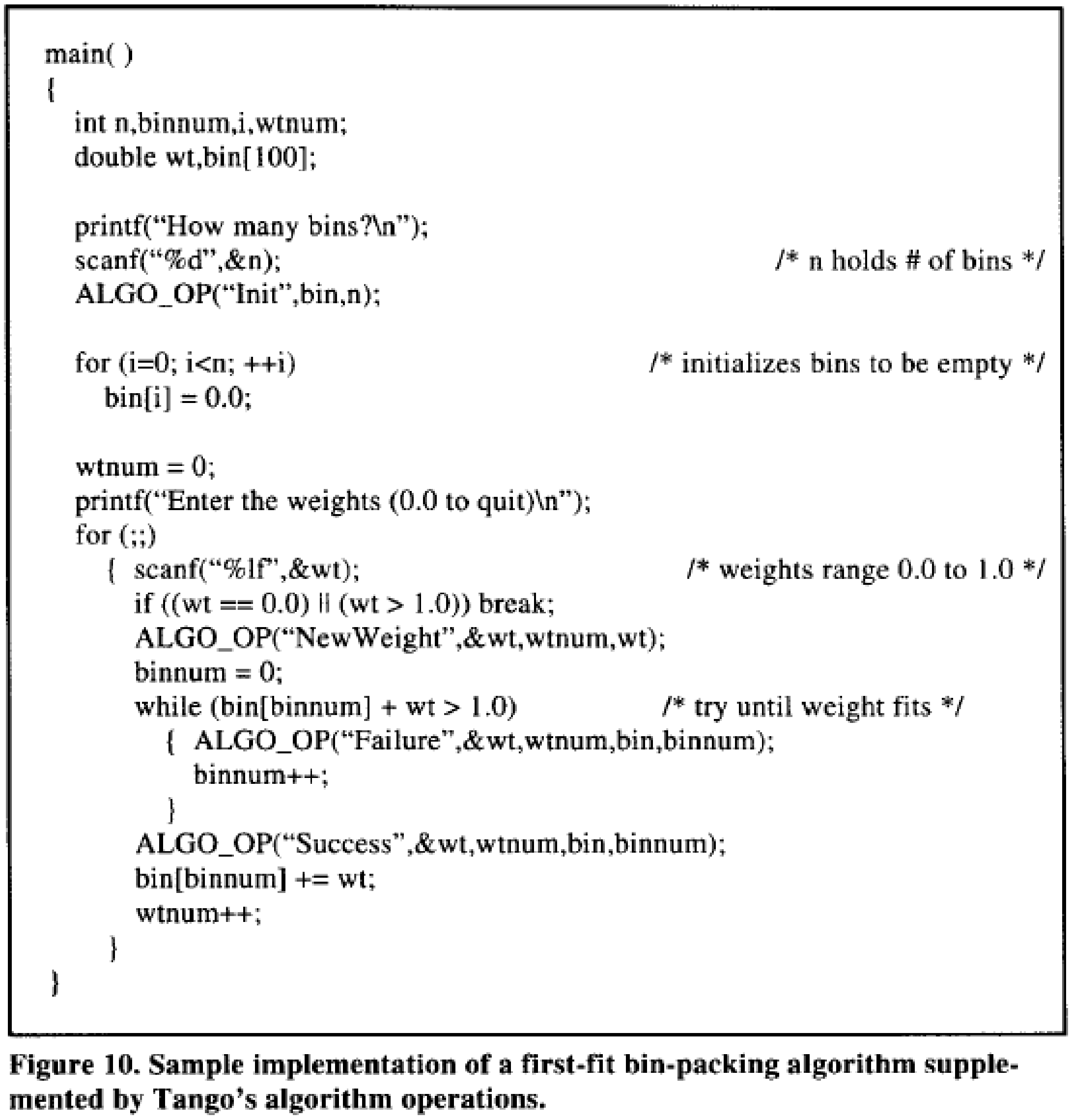
Currently there are several accepted design architectures for animation systems. There are three broad categories, pattern-based systems, command language systems, and history-based systems. These divisions are predicated on what specification technique a system uses. Pattern-based systems use design patterns, command language systems generate scripts that are later compiled, and history-based systems store information describing the state of their code at different times during execution.

Pattern-based systems use some programming design pattern to translate code into graphics. These systems are relatively scarce. The Data Structures Library in Java (JDSL) [9, 10, 40] is an event-driven system that uses the template methods pattern to recognize interesting events. The template method pattern uses abstract algorithm classes that define the order of operations, but not the function of each operation. In this way the structure of the algorithm is enforced while behavior is customizable [7]. Algorithma99 separates itself into components based on object-oriented best practices [4]. Automatic algorithm generation of arrays can be achieved using C++ templates [24] but is not generalizable to other data types or languages. Though many patterns for sorting are known [23], they are used infrequently in algorithm animation system design.

History-based systems are more common than pattern-based systems. Systems that keep history lend themselves quite easily to a state-driven specification technique. One of the earliest, Zstep [16], kept a full history of all states that data had been in. While this was space-inefficient, it allowed for one of the earliest full undo-redo models in algorithm animation. Currently systems like ANIMAL [30, 31] keep only a subset of the history, called animation steps, which can be iterated through.

The most common system type is command language. These systems generate scripts that are then compiled into some type of animation frame. Many of these systems have the unique benefit of being able to import and export scripts, as well as being able to translate between scripts. This allows scripts generated by one system to be animated by a different system. A proliferation of command languages hinders this effort. While command languages are commonly used, it is notoriously difficult to achieve automatic animation using only a command language.

There are many command language generating systems. The most well-known are JAWAA [27], SAMBA [38], TANGO [39], POLKA [37], and LEONARDO [5]. The method of command language generation varies between systems. Some annotate the code explicitly, such that there are non-executable comments labeling the code that are then converted into animation commands embedded in a script. JAWAA, SAMBA, TANGO, and LEONARDO take this approach.



```
main( )
{
   int n,binnum,i,wtnum;
   double wt,bin[100];

   printf("How many bins?\n");
   scanf("%d",&n);                                   /* n holds # of bins */
   ALGO_OP("Init",bin,n);

   for (i=0; i<n; ++i)                      /* initializes bins to be empty */
      bin[i] = 0.0;

   wtnum = 0;
   printf("Enter the weights (0.0 to quit)\n");
   for (;;)
      {  scanf("%lf",&wt);                       /* weights range 0.0 to 1.0 */
         if ((wt == 0.0) || (wt > 1.0)) break;
         ALGO_OP("NewWeight",&wt,wtnum,wt);
         binnum = 0;
         while (bin[binnum] + wt > 1.0)        /* try until weight fits */
            {  ALGO_OP("Failure",&wt,wtnum,bin,binnum);
               binnum++;
            }
         ALGO_OP("Success",&wt,wtnum,bin,binnum);
         bin[binnum] += wt;
         wtnum++;
      }
}
```

Figure 10. Sample implementation of a first-fit bin-packing algorithm supplemented by Tango's algorithm operations.

Figure 1. Typical Intermediate Language Generation [39]

API calls are another common command language generator. Special method calls are interspersed within the programmer's code to map data operations to command language. Figure 1 shows a bin packing algorithm with ALGO_OP commands in TANGO. API calls define certain methods as important methods—when they are called they append to a script. POLKA and XTANGO both use this approach [37]. By definition, API calls are interspersed with programmer code, and so programmers must be aware of the animation. This restricts the possibility of automatic animation generation when using API calls.

3. Event Based Patterns

We designed our system guided by the principle of separation of concern. State-driven animation is often achieved using a mapping before the algorithm is run. This means that some sort of annotation or script must be supplied in order for the animation to be visualized. This disallows run-time changes to animation and creates unwanted dependencies between the algorithm and the animation. Visual programming also will not suit our purposes; indeed it is more suited to visualizing programs than visualizing algorithms because it directly shows commands and variables.

This leaves us with event-driven and automatic animation. Automatic animation is by far the more complex. We will first provide an event-driven engine that fits our goals, and then use this engine to attain pseudo-automatic animation.

One of the first things to note is that event-driven animation is concerned with responses to “interesting events” [15]. Conventionally animations of this type are created by identifying these interesting events and then annotating or making special method calls in the code where they occur. This presents the same problem as state-driven approaches: it forces the coupling of algorithm and animation. Additionally, it discourages the creation (or modification) of algorithms because it necessitates new annotations to be added upon each code change.

3.1 Algorithm Decomposition

Instead of concerning ourselves with interesting events in the algorithm [9, 10, 40], we propose identifying interesting events in the data structure itself. For instance, if we have an algorithm that randomizes a vector, we could do one of two things. We could write an annotation every time an element is changed or when elements are swapped. Alternatively, we could define, as part of the data structure, that the `void swap(Element e1, Element e2)` and the `void set(int index, Element newVal)` are interesting events. We could go further, and assume that all methods that change a data structure and are in the interface of that data structure are interesting events. For a simple list this might mean we are interested in the following methods:

```
void swap(int index1, int index2);

boolean add(Element e1);

boolean remove(Element e1);

void set(int arg0, Element arg1);
```

Figure 2. Interesting Events

Merely identifying interesting operations on a data structure does not allow us to map them to an animation. Given the event-based nature of our approach, using the observer pattern [7] seems natural. If we let the data structure we are interested in be observable, then every interesting operation fires an event which can be caught and handled by any observer. This event generation is built into the data structure's methods, and so is transparent to the algorithm programmer who can perform any number of operations on the data without ever knowing that events are being fired underneath.

3.2 Modifying the Observer Pattern

Conventionally, the observer pattern is only concerned with firing events for changes to a data structure. While this supports some algorithms, many algorithms have important steps that do not change the data involved. Visiting nodes while searching for a least-cost path in a graph, for instance, is very important. Each visit to a node should be animated, yet the node itself is not changed.

Therefore we must modify the conventional observer pattern. The first change we introduce is making reads optionally observable. Thus if a programmer needs to catch events for node accesses, he/she is able to. We make read events optional for the sake of efficiency. Reads are a common operation and we do not want to fire events for each read unless the programmer decides they are necessary to animate a certain algorithm.

4. Command Architecture

While firing events when important operations take place separates the animation from the data and algorithm, it is not sufficient to meet some of the animation goals we set above. For instance, if every event is immediately caught and translated into a graphical change, then each data update will immediately be animated. This breaks our constraint that users should be able to control when an operation is animated and at what speed it is animated. Further, event handling provides no inherent means to undo, much less redo, an operation.

Because event patterns can support command language generation, they are more versatile than command languages. It is entirely feasible that events could be handled by merely writing intermediate language to a file. If this were done the intermediate language generation would be separate from the data and algorithm. Assume a user wants to sort a vector, requiring four swaps. Each of these swap events would be handled by an observer that adds the script to graphically swap two elements to the end of a file. This file could be run through a script compiler to generate animation frames once the algorithm had completed. Because this follows the principle of separation of concern, it is a step in the correct direction, but it still requires the algorithm to run to completion before it can be animated, and therefore it does not support run-time animation.

4.1 Command Objects

If we were to use the approach above, the intermediate language and the frames it would generate are a sort of hard-coded history of what has happened. We could step through each frame and step backwards to undo. Instead of having a static execution history, however, we can introduce a history that is modified as the algorithm runs. Instead of writing an intermediate language to a file, we instead add command objects [7] to a history that is shared between several threads.

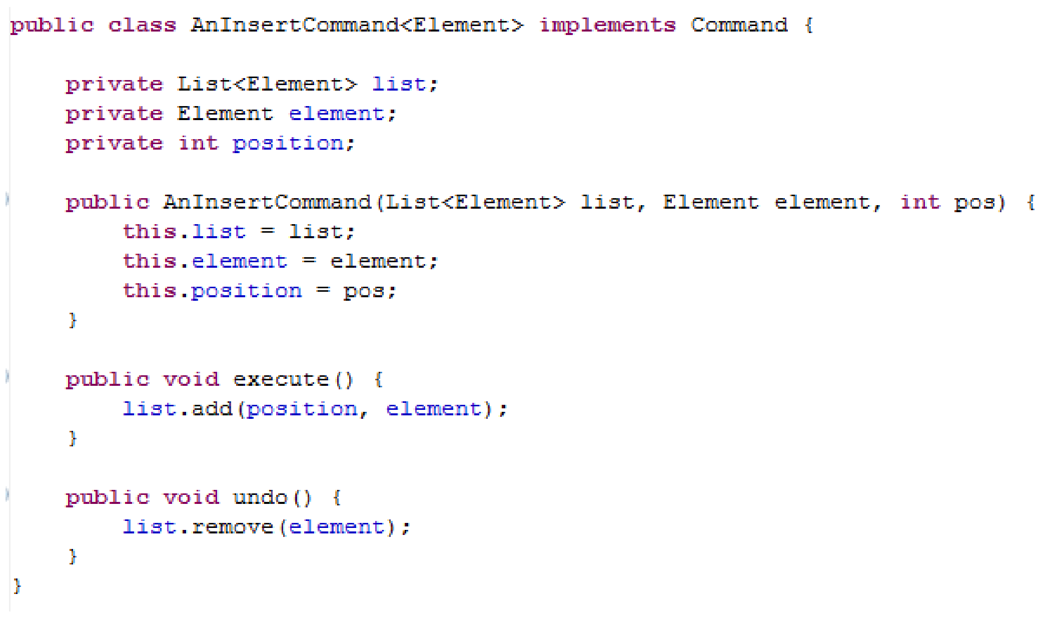
The command object design pattern embeds an operation, along with its parameters, inside an object [7]. In this way it becomes possible to defer the execution of the encapsulated operation until a later time. Now every time that an operation is performed on the data, a command object that corresponds to the operation is created. This command object is then added to a history containing all command objects created for the data. This history of command objects can then be used to step through the animation.

Assume we have the same four swaps as before. Now there is a history of four command objects, each associated with a vector, a swap operation, and the two indices to swap. Instead of animating changes every time an event is fired, we now animate changes only when the corresponding command object is executed.

We have done no research on intuitive or optimal user interfaces. However, many commonly used GUIs have previous and next buttons; we assume this is the case. If the user controlling the animation were to hit the next button, then the next un-executed command object in the history would be executed, causing the animation for that specific operation to occur. This process is explained in greater detail in section five.

4.2 Inverse Operations

A history of command objects allows for delayed execution of animation, but does not directly allow for undoing animated operations. To facilitate the undoing of these operations it is necessary that each command object be an undoable command object, that is, have an undo operation that corresponds to its execute operation [7]. This undo operation is also stored inside the command object and is the logical inverse of the execute operation.



```
public class AnInsertCommand<Element> implements Command {

    private List<Element> list;
    private Element element;
    private int position;

    public AnInsertCommand(List<Element> list, Element element, int pos) {
        this.list = list;
        this.element = element;
        this.position = pos;
    }

    public void execute() {
        list.add(position, element);
    }

    public void undo() {
        list.remove(element);
    }
}
```

Figure 3. An Insert Command Object

We can see an example of an undoable command in Figure 3. When this command is executed it will insert an element into a list and when it is undone it will remove that same element from the list.

Now assume, for a moment, that a user/programmer interested in viewing a sort algorithm animated runs the sort algorithm, generating the same four swaps as before. Now instead of the animation corresponding to the actual algorithm, the animation corresponds to our history of commands. When the user wants to see the next step in the animation (this can be thought of as the next frame) he or she indicates this desire by incrementing the current index in the history of command objects and executing the corresponding command object. If the user decides to undo the operation, or reverse the animation step, then he or she merely decrements the pointer in the history of command objects and calls the undo method on that command object.

Although this is a classic undo-redo model using command objects, it provides the unique benefit to animation systems of being able to change as the algorithm runs.

Suppose that our four-swap sorting algorithm only gets halfway through running when the user decides to interact with the animation. At this point only two of the four command objects have been created. Despite the fact that the algorithm is currently running, the user is still able to execute and undo those command objects; he or she can view every part of the algorithm that has completed up until that point. Further, suppose the program runs a sorting algorithm, waits for some user input, and then runs a randomization algorithm. YAAS can animate (both forwards and backwards) the sorting portion of the program while waiting for user input. Once the user decided to continue and the randomization took place, the history of command objects would contain all the commands for sorting in addition to all the commands for randomizing, allowing the animation viewer to view both algorithms back-to-back.

Up until this point we have left out details about how the event-handling and command object components of YAAS are connected and ultimately translated into graphical animations. The connection between different components is covered in detail in section six.

We have also intentionally omitted discussion of the mapping from the execution of a command object to the animation displayed on screen. This mapping is covered in detail in sections six and seven. It is worth noting that in order for the execute-undo model to work graphically it is necessary that the graphical changes made when a command is executed must have an inverse.

4.3 Comparison to Other Systems

Using command objects it is possible to animate any algorithm on a data structure merely by knowing how to render each operation available on that structure. This means that once a mapping from command object execution to animation is made, then a user who wants to see his or her algorithm animated merely has to register the data structure with YAAS and then write their algorithm.

As a teaching tool, the flexibility provided is useful. If an instructor is interested in a student's conceptual knowledge of an algorithm he/she can require them to construct the mapping from command execution to graphical layout. If the instructor is more concerned with algorithm correctness, then he or she can provide the mapping, and let the students write the algorithm.

Systems like Jeliot and JSamba [21, 38] that use intermediate language generation are more tightly coupled than our model. They often require annotations or API calls alongside the algorithm and do not support concurrently running algorithms and animations.

Systems like JAWAA [27] are not limited to responding to data structures. While they can be useful in allowing students a way to create exciting projects in a programming course, they require animation text to be written to script files which are later read. Further, there is no undo-redo support. This breaks the cognitive-constructivism model [13] and so is not ideal for teaching about data structures or algorithms.

Run-time algorithm animations are generated by JDSL [9, 10, 40]. Instead of providing the option of writing algorithms; however, JDSL uses the template method pattern [7]. This is a useful paradigm, but it is more suited for teaching data structures than algorithms and does not allow for user-defined algorithms. Further, much of the graphical information is coupled alongside the data.

In the next section we will investigate how our data is decoupled from its graphical animation. We discuss the path an event follows, from the original operation in the user's data, through several intermediate event buffers, and ultimately to the change in the logical structure of a final data structure.

5. Undo-Redo Engine

The path an event takes from creation through animation is full of indirection, following a key idea in software development. The event must eventually be animated. YAAS uses a GUI-generating package called ObjectEditor [6], covered in more detail in section eight, which necessitates that graphical changes correspond to logical changes in a class' structure. This means that in order for a shape to be added or changed in our GUI, the shape (or a logical structure corresponding to that shape) must be added or changed in a class that ObjectEditor displays graphically, or 'edits.' It follows that we need some type of structure containing all these shapes; for now we will use a list that we will refer to as the 'visualizer'. This visualizer will be edited by ObjectEditor and each shape in the resulting GUI will correspond logically to an object stored in the visualizer. If an object stored in the visualizer changes logically, i.e. a shape's x-component is increased, then the same change will occur graphically in the GUI. It is important to note that YAAS is not coupled to ObjectEditor. Any toolkit or GUI generator can inspect the visualizer and display the list of shapes; we have chosen to use ObjectEditor in our development cycle for ease of use and familiarity with the tool.

It would be simple to have a command’s execution change a shape in the visualizer. However, this requires a command to have access to the visualizer, and to know about what shape needs to be added. If we want YAAS to support any type of animation then we must separate command execution from the visualizer.

5.1 Command Buffers

We separated our original data from the graphical view using the observer pattern, and we can use it again here. Instead of actively changing anything when a command is executed, we merely modify a copy of the original data. We call this copy of the original data structure the command buffer. When the original data is changed an event is fired that is caught and transformed into a command object that is a copy of the operation but with a different object as the target. When this command object is executed or undone it performs the same operation—with the same argument—that was conducted on the original data, but on the buffer. The buffer acting as both an observer and an observable fires an event, which can be caught and handled by another observer, including the visualizer.

5.2 Intermediary Buffer Layers

To simplify the programming, the class that handles the data’s events can be the same class that manages the history of command objects.

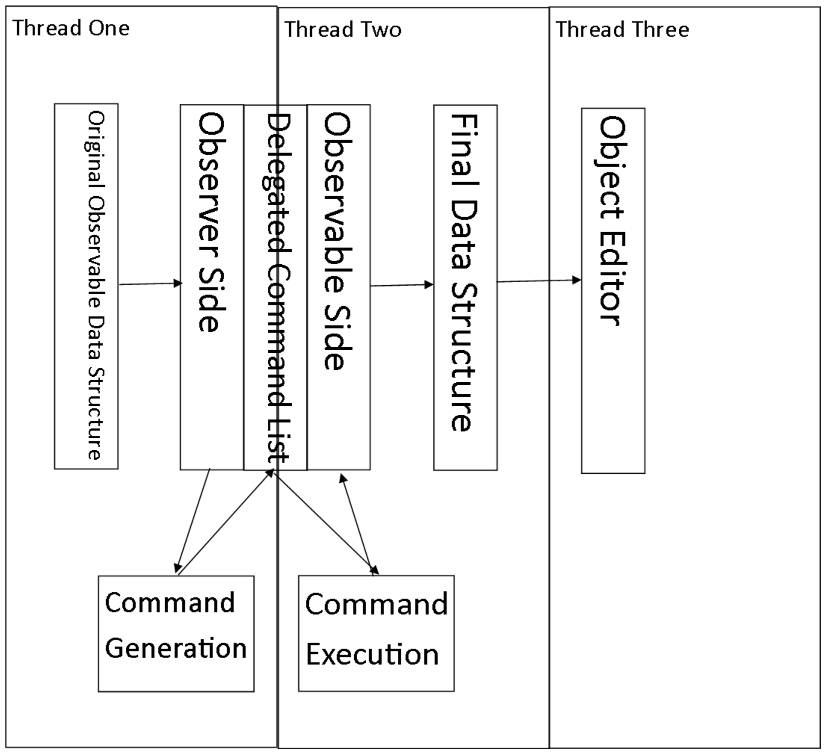


Figure 4. Net Effect of an Operation

Figure 4, above, shows the effect that a change in the data has across our system. The operation immediately takes effect on the original data. Once the data has been successfully changed, an event is fired. This event is caught by all observers, one of which is the command buffer that is both an observable and an observer. This buffer delegates a history of command objects. When the buffer catches an event, it creates a command object that represents that event, but sets itself as the target. So when two indexes, i_1 and i_2 are swapped in the original data, an event is fired that results in the creation of a command object that, when executed, swaps the same indices in the buffer. This command can be executed programmatically or from a GUI interface. Once this command has been executed, the buffer fires an event signifying that it has changed. All observers of the buffer, including the visualizer, catch and handle this event. The visualizer updates its history of shapes accordingly, which is reflected in the GUI.

There are two fundamental reasons we use a command buffer: modularity and asynchronicity. Had we decided that every change to the original data immediately corresponded to a change in the final data then there would be no method to alter the nature of the fired event. Suppose, for example that two elements are swapped during the execution of our sorting algorithm. If these changes immediately take effect, then two elements are also swapped in the visualizer, but we may want this swap to be animated at a lower level, as the creation of a temporary variable and two variable assignments. In this case, we must catch the event fired from the original data and transform it into two new events that will ultimately be processed by the visualizer. By introducing trappers, or data copies that are both observable and observers, we allow events to be caught, transformed, and then fired. In this sense, the visualizer knows that it is receiving an event from a trapper and that this event was caused initially by a change in the original data. In the same way, the original data fires an event that is caught by a trapper and ultimately received by the visualizer. A more accurate diagram is shown in Figure 5, below. The trapper structure is described in detail in section seven.

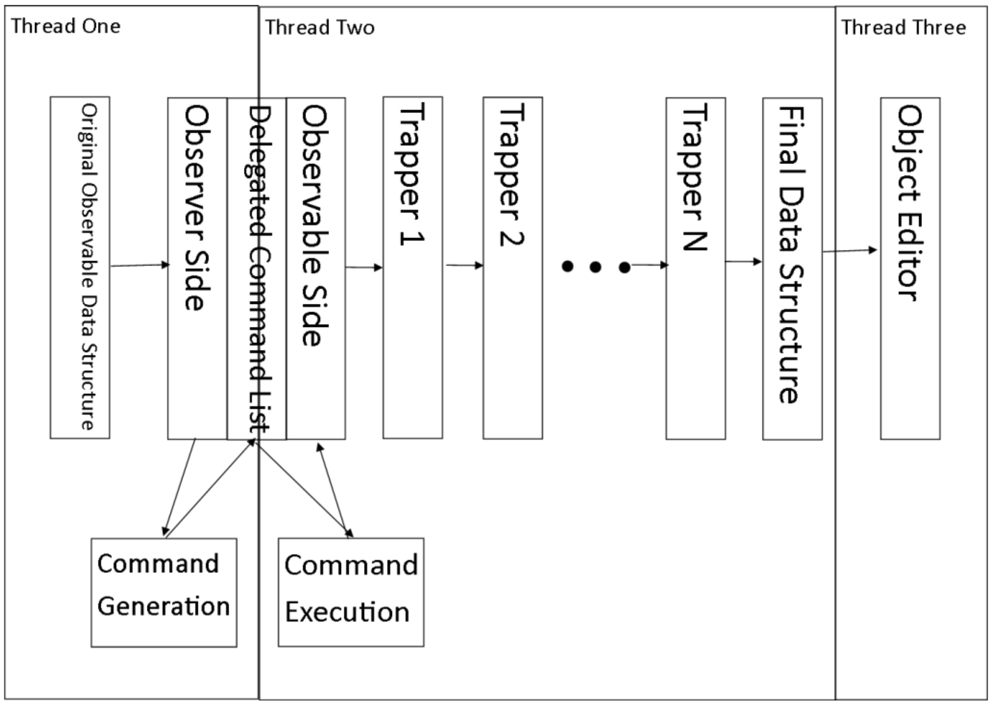


Figure 5. Event Path with Trappers

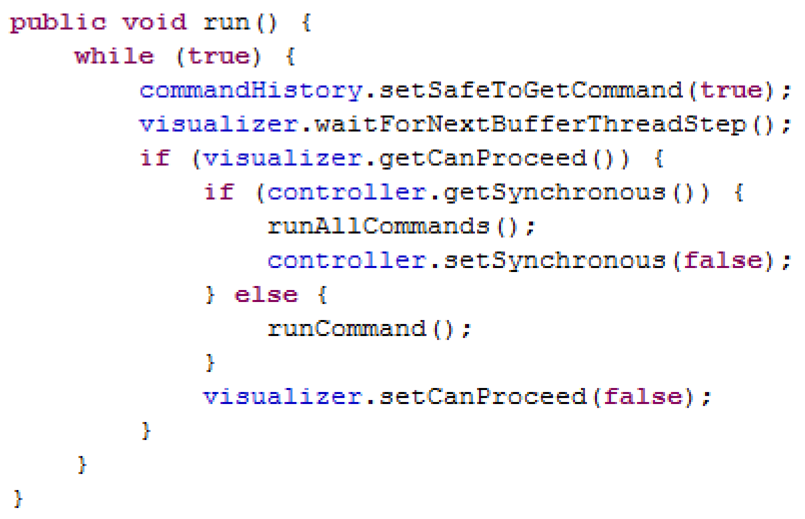
Currently YAAS supports several data patterns. We provide support for Java's Observer/Observable pattern, bean patterns, and vector patterns. If the programmer's data follow any of these common conventions then it is already supported by YAAS. Otherwise, we support programmer-defined data and patterns. If the programmer decides on this course of action they must write their own command objects and command buffer.

5.3 Command Synchronization

We noted earlier that if commands were immediately executed then modifications to the GUI would show up immediately; the system would be synchronous. It would be better to provide the option of running synchronously but also allow users to block command execution until they want to see it graphically. Animation viewers should be able to choose whether they want to view a synchronous or asynchronous animation.

With this end in mind, the command buffer allows a place for commands to be stored that have not yet been executed. To enable synchronous animation we execute commands as soon as they are created. To provide asynchronous animation we block command execution until some condition becomes true.

In the case of asynchronous command execution we have a loop that waits for input from the animation viewer. This loop blocks until three conditions become true: the command before this one has completed execution, the animation viewer indicates that they want the next phase of the animation to complete, and there are more commands to execute. This is shown in Figure 6 below.



```
public void run() {
    while (true) {
        commandHistory.setSafeToGetCommand(true);
        visualizer.waitForNextBufferThreadStep();
        if (visualizer.getCanProceed()) {
            if (controller.getSynchronous()) {
                runAllCommands();
                controller.setSynchronous(false);
            } else {
                runCommand();
            }
            visualizer.setCanProceed(false);
        }
    }
}
```

Figure 6. Synchronization Loop

The thread waits until it is notified by the visualizer that it can continue. In asynchronous mode this occurs when the animation viewer indicates that he or she wants to continue by invoking a ‘next’ method in a class that controls the viewed animation. In synchronous mode this notification occurs as soon as the visualizer has completed the last command. We impose the limitation that only one command can be executing at any given time. This prevents concurrency errors that could arise from multiple operations changing the same data at the same time.

The history of command objects in the buffer is treated as if is iterable. It is necessary to check that there are command objects available to execute before attempting to execute one. Commands are also ordered; they must be executed in a first-in-first-out manner. The only complication is when undo is introduced. If the user decides to undo a command, then the direction in which the command history is being iterated over reverses.

It is also worth noting that once a command object has been created and added to the history inside the buffer, it should not be removed. In this way the command object history is a permanent record of every operation that has taken place on the user's data. We can iterate over all states the original data has been in.

5.4 Comparison to Current Undo-Redo Methods

Other systems use one of several methods to perform undo-redo. Some systems like JHAVE [21], AlgAE [40], and SAMBA [37] step through scripts that are written in a system-specific algorithm visualization languages. ZStep [15] saves a complete history of every path the animation has taken and reloads previous states to undo operations. LEONARDO [4] contains an entire operating system, and uses a virtual CPU combined with its own animation language to perform undo-redo. Other systems may use a decorator pattern on the algorithm itself to allow undo redo [5, 8, 9, 39], or a series of linked animations [29].

Those systems that use scripts cannot respond to changes in data at run-time; the script must be generated and then the animation is set in stone. These systems do provide the unique ability to import and export (and even translate between) different animation scripts and reuse previously generated animations, even from other systems. However; the nature of scripts is such that automatic animation cannot be achieved.

Saving a complete history wastes space. Our system, instead of saving a complete history of all forms the original data has ever been in, saves a history of operations which can reproduce the form of the original data at any point in time. When a command is executed the buffer is the same as the original data was at the time of command creation.

Using decorator patterns in the algorithm itself is good solution; however, it assumes that a programmer will be programming data structures, not algorithms. Our system allows the programmer to design algorithms and define their own data structures, employing a reasonable amount of work.

ANIMAL's method of linking animation steps [30, 31] couples graphical information with data, and so is not optimal. Of the above methods, scripts, decorations, and linked animation steps all couple graphical information with the data itself. Our undo-redo model is independent of the graphical system; if a programmer wanted, he or she could use the engine for a completely different purpose and never know it was designed for animation.

6. Event Handling

Now that the undo-redo structure of command execution is clear we can take a closer look at how events are transformed as they travel from the original data to the visualizer. Any number of trappers—data copies that are both observable and observers—can be inserted between the buffer and the visualizer. The visualizer contains a trapper manager, a class that is responsible for the management of all trappers. A trapper can be added to or removed from the end of this chain. An event is processed by the buffer, all trappers, and then the visualizer. If there are no trappers, then the event goes straight from the buffer to the visualizer.

6.1 Trapper Layers

Unlike the buffer, each trapper has direct access to the visualizer. This means that in addition to transforming events, they can modify the shapes contained in the visualizer in order to directly change the GUI. Let's look at our earlier example of a swap command that we want to transform into the creation of a temp variable and two set commands. It is simple to catch the swap event and create two new set events. We do this by catching the swap event, reading the values to be switched, and then setting those values in the current trapper accordingly. These sets then fire new events to be caught by the next trapper in the chain, as seen in Figure 7 below.

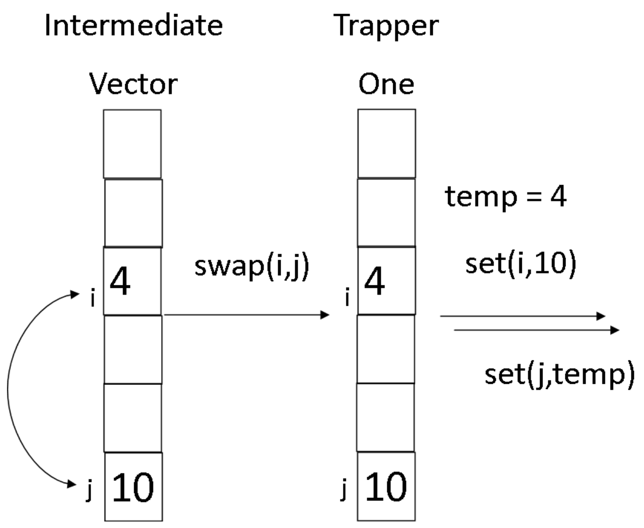


Figure 7. Event Transformation

Handling the assignment of the temporary variable is more problematic, because it doesn’t directly change the data and so no event will be generated. This is why it is necessary for trappers to have direct access to the Visualizer; a trapper may need to change a GUI element for an operation that doesn’t correspond to an event performed on the data.

6.2 Net Effect of an Operation

It is now possible to trace the entire life cycle of an event. A user or programmer changes the data, causing an event to be fired. This event is caught by the buffer and a command object is created. At some later time this command object is executed, causing a change in the structure of the buffer. This change causes the buffer to fire an event. This event is caught and re-fired by any number of trappers that may change the GUI or transform the event. Finally, the last trapper’s event is caught and handled by the visualizer. The internal structure of the visualizer determines how the animation will be displayed.

6.3 Automatic Animation

It is now possible to see how automatic animation can be attained. Assume that either the programmer or developer has created trappers for a list-based data structure. These trappers control how an animation will be displayed. For instance, when an add event is received, a new box with the corresponding element flies onto the GUI in the correct position (this process will be covered in detail in section eight). Equivalent animations are programmed for swaps, removals, and other common list operations.

Now we pose the question, what does a programmer have to do to create an animation of an algorithm? All of the animation code is separate from the algorithm, and all of it works with any operations on an observable list, not with a specific algorithm. The programmer merely has to write an algorithm using an observable list, and the animation will be generated with no other programmer input.

Pure automatic animation imposes no restrictions on the programmer and is notoriously difficult. We impose the restriction that the programmer's data must be observable. This is not a heavy restriction; it is often trivial to change a data structure to be observable. Additionally, there is a fixed cost: trappers and buffers must be written. After this fixed cost, animation is achieved automatically. YAAS's pattern is more general than previous means of automatic animation generation, which often require language support, such as with automatic generation of array animations that necessitate C++ templates to function [23].

7. Graphics

It is worth noting that the architecture described thus far is completely independent from any sort of graphical display. Additionally, it is independent from both the data structure and the algorithm being used. In this sense, our undo-redo engine could be used with any graphics display, or even for an application that has nothing to do with graphics. The visualizer is ultimately a list of shapes. Each element has x and y coordinates, height and width, and descriptive attributes such as color, angle, photo, or text. Any system that can generate GUIs out of shapes could be used to animate the visualizer. We have chosen to use a GUI toolkit called ObjectEditor [6]. We find integrating our system with this toolkit useful for three primary reasons: the relative lack of support for animation in user interface toolkits [11], simple translation from a program's logical structure to graphical displays, and prior experience with the toolkit.

YAAS provides, through ObjectEditor, access to common graphical shapes, images, textual objects, etc. In addition, it handles any amalgamation of simple shapes that can be treated as a whole unit through x, y, height, and width properties. In addition to predefined shapes, we support shape patterns. If a class has a common shape name and implements common shape properties ObjectEditor will automatically generate a GUI representation of the shape. This allows programmers to use shapes they have implemented themselves.

7.1 Object Editor Uses Logical Structure

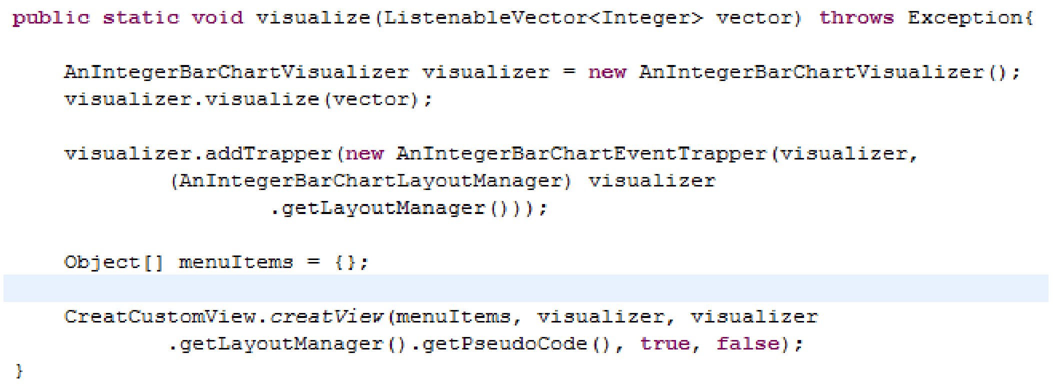
ObjectEditor uses introspection to determine a class' logical structure and then displays it graphically. If it sees a class contains an object that has an x and y coordinate, width, and a common shape name then it will display that shape with the corresponding attributes. This made it simple for us to translate the visualizer into a display; we merely needed it to contain a list of objects that could be interpreted as shapes. To animate changes we change the attributes of each logical shape inside the visualizer by small amounts over time. ObjectEditor's display automatically catches these changes and updates itself accordingly.

7.2 Graphical Layout Management

Up until this point we have assumed that there was an intrinsic mapping from the original data to the data that is displayed in the GUI. To provide the widest support for varying graphical representations we allow the programmer to decide how to display the animation. The visualizer has an instance of a `LayoutManager<UserDataType>`, or an interface that provides a mapping from data to shapes. This interface has one method, `ListenableVector<Shape> display(UserDataType v)` that translates from the user's data to a list of shapes. This method is run once, when the original data is registered with the visualizer. Each trapper then modifies this list of shapes based off of the events it receives.

7.3 API

The code required to animate a vector is below: First the visualizer is instantiated. It then visualizes the user's observable vector.



```
public static void visualize(ListenableVector<Integer> vector) throws Exception{

    AnIntegerBarChartVisualizer visualizer = new AnIntegerBarChartVisualizer();
    visualizer.visualize(vector);

    visualizer.addTrapper(new AnIntegerBarChartEventTrapper(visualizer,
            (AnIntegerBarChartLayoutManager) visualizer
                    .getLayoutManager()));

    Object[] menuItems = {};

    CreatCustomView.creatView(menuItems, visualizer, visualizer
            .getLayoutManager().getPseudoCode(), true, false);
}
```

Figure 8. Visualization Code

A trapper is added to process events. Finally, in order to display the animation, we register the visualizer with the ObjectEditor toolkit by creating an array of menu items and calling a method to create the view. If a separate GUI was used, the menuItems and createView lines could be omitted. The command object creation, synchronization, and undo-redo engine is created under the scenes; the programmer need not know about its implementation.

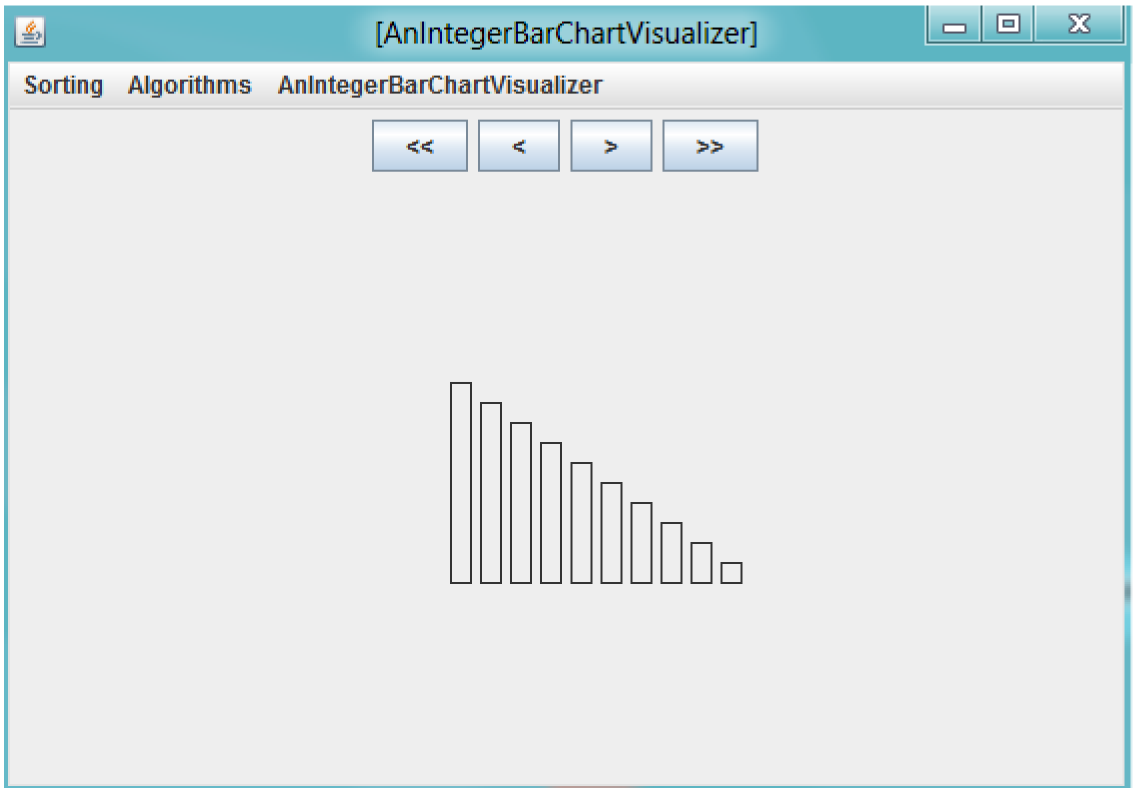


Figure 9. A Sorting Animation Visualized With ObjectEditor

7.4 Comparison to Current Graphical Systems

The graphical capabilities of ObjectEditor suit our needs. It includes common GUI components without necessitating window management. In this way it is similar to the graphical capabilities offered by other animation systems. Our work focuses primarily on the internal manipulation of data, and so we have made no attempt to mirror systems like ANIMAL [30, 31] that allow animation creation via GUI editing.

8. Critique of YAAS

In comparison to other algorithm animation systems ours is quite young. Many systems have had teams working for decades to provide a broad base of algorithm support and GUI customizability.

While the concept of our undo-redo engine works for any observable data structure, the current implementation provides support only for data that conforms to common patterns: beans, observables, and lists.

Because we do not use any sort of scripting language, we do not support the importation or exportation of other tool’s animation scripts. Many intermediate language systems provide these utilities. We can produce a script using our event pattern, but cannot translate from an animation to a script, or load a script. Additionally, we cannot yet convert our animations into RDF and so cannot embed them into PowerPoint presentations, a primary medium for course lectures.

Because the logical structure of the visualizer has no access to the original data (or program containing that data) it is difficult to include source code and source code highlighting. It is possible to include text, and highlight text, which allows the capability to include source code resources. However, there is not a simple way to retrieve the actual source code for the algorithm in question. Instead the programmer must provide pseudo code text and a mapping from that text to command execution.

8.1 Areas for Future Work

There are several areas of study that we would like to pursue in the future. We would like to generate a wider range of views. This means creating various new trappers and layout managers.

We feel it is important to perform empirical studies on how algorithm animation systems are used by programmers. Many studies have been performed on their effectiveness as a teaching aid [12]; however, few studies analyze how effective different tools are at providing simple development of new animations. We hypothesize that decoupling components and automatic animation generation will simplify the creation of animations as well as reinforce good programming style. It is important to test this hypothesis.

9. Conclusion

There are many fine algorithm animation systems. They range from source code animators to debugging tools, from presentation generators like PowerPoint to operating systems with their own virtual CPU, from simple stick figure animations to videos. Despite such a breadth of systems, GUI designs, and underlying architectures, there are few methods of translating from algorithm to animation.

The most common method is by using some type of command language; these systems run an algorithm which generates a script file that is later processed by a compiler that generates frames. These frames are then iterated over. Other systems save a history of program states. This is not space efficient, but does allow undo-redo. Improving upon this idea, there are systems that link animation steps, as opposed to the state of the data. There are a few systems that use design patterns to indicate important events in the algorithm in question and then respond to these events. We have developed an approach similar to this method, but with the focus on interesting events in the data as opposed to those in the algorithm.

We were able to decouple the operations performed on an observable data structure from the animation of that data. By mapping operations to undoable command objects, we allowed animations to run concurrently alongside the data structure’s operations. We built an engine composed of observable-observing data copies to process events and provide a logical structure that is simple to convert to a GUI.

We noted several important components of animations systems and added a few of our own. Because YAAS was designed using Java, it is platform independent. It has support for inclusion of pseudo code and highlighting through inclusion of textual and visual objects in the visualizer. YAAS is not dependent on connection to any network once it has been installed. We are able to run animations both synchronously and asynchronously. It provides support for any algorithm that can be run on an observable data structure.

In addition to these conventional requirements, we allow an animation to be viewed while the corresponding algorithm is being run. We allow the programmer to define how to display graphics, have complete separation of concern between logic and graphics, and ultimately provide support for automatically generated animations from any algorithm run on an observable data structure.

10. References

[1] Rodger, S. (n.d.). Using Hands-On Visualizations to Teach Computer Science.

[2] Baker, R. S., Boilen, M., Goodrich, T. M., Tamassia, R., & Stibel, A. B. (n.d.). Testers and Visualizers for Teaching Data Structures.

[3] Concepcion, A. I., Leach, N., & Knight, A., (2000) Algorithma 99: an experiment in reusability & component based software engineering. 31st ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE2000), Austin, TX, 162-166.

[4] Crescenzi, P., Demetrescu, C., Finocchi, I. & Petreschi, R., (2000) Reversible execution and visualization of programs with LEONARDO. Journal of Visual Languages and Computing 11, 125-150.

[5] Dershem, H. L., & Brummund, P., (1998) Tools for Web-Based Sorting Animations. 29th ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '98), Atlanta, GA, 222-226.

[6] Dewan, P. (n.d.). ObjectEditor. Retrieved from Prasun Dewan's Home Page: http://www.cs.unc.edu/~dewan/oe

[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J., (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Oxford University Press, Inc.

[8] Gelfand, N., Goodrich, T. M., & Tamassia, R. (n.d.). Teaching Data Structure Design Patterns.

[9] Goodrich, T. M., & Kloss II, G. J. (n.d.). Tiered Vector: An Efficient Dynamic Array for JDSL.

[10] Goodrich, T. M., Handy, M., Hudson, B., & Tamassia, R. (n.d.). Accessing the Interal Organization of Data Structures in the JDSL Library.

[11] Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Terasvirta, T., & Vanninen, P. (1997). Animation of User Algorithms on the Web. IEEE Symposium on Visual Languages, (pp. 360-367).

[12] Hudson, E. S., & Stasko, T. J. (1993). Animation Support in a User Interface Toolkit: Flexible, Robust, and Reusable Abstractions.

[13] Hundhausen, C., Douglas, S., & Stasko, J. (2002). A Meta-Study of Algorithm Visualization Effectiveness. Journal of Visual Languages and Computing, 259-290.

[14] Karavirta V, Korhonen A, Malmi L, Naps T. 2010. A comprehensive taxonomy of algorithm animation languages. Journal of Visual Languages and Computing 21:1-22.

[15] Kerren, A., & Stasko, J. (2002). Algorithm Animation. Software Visualization State of the Art Survey, 1-15.

[16] Lieberman, H. & Fry, C. (1998) ZStep 95: A reversible, animated source code stepper. In: Software Visualization (J. Stasko, J. Domingue, M. H. Brown & B. A. Price, eds). MIT Press, Cambridge, MA, Chapter 19, 277-292.

[17] Lieberman, H., Reversible Object-Oriented Interpreters, First European Conference on Object-Oriented Programming, Paris, France, Springer-Verlag, 1987.

[18] Mak, K. (n.d.). Graph Animations with Combinatorica. Retrieved 8 30, 2011, from Skiena's Home Page: http://www.cs.sunysb.edu/~skiena/combinatorica/animations

[19] Martin, D. (n.d.). Sorting Algorithm Animations. Retrieved 8 30, 2011, from http://www.sorting-algorithms.com

[20] Morris, J. (1998). Animated Algorithms. Retrieved 8 30, 2011, from John Morris' Home Page: http://www.cs.auckland.ac.nz/~jmor159/PLDS210/alg_anim.html

[21] Myller, N. (2004, February 2). The Fundamental Design Issues of Jeliot 3. University of Joensuu Master's Thesis.

[22] Naps, T., Furcy, D., Girssom, S., & McNally, M. (2008, 6 21). Java Hosted Algorithm Visualization Environment. Retrieved 8 30, 2011, from http://jhave.org

[23] Nguyen, D., & Wong, S. B., (2001) Design patterns for sorting. 32nd ACM SIGCSE Technical Symposiumon Computer Science Education (SIGCSE2001), Charlotte, NC, pp. 263-267.

[24] Rasala, R., (1999) Automatic array algorithm animation in C++. 30th ACM SIGCSE Technical Symposiumon Computer Science Education (SIGCSE '99), New Orleans, LA, pp. 257-260.

[25] Rodger, S. (2002). Introducing Computer Science Through Animation and Virtual Worlds. Thirty-third SIGCSE Technical Symposium on Computer Science Education, (pp. 186-190).

[26] Rodger, S. H. (n.d.). Using Hands-On Visualizations to Teach Computer Science.

[27] Rodger, S., & Pierson, W. (1998). Web-based Animation of Data Structures using JAWAA. Twenty-ninth SIGCSE Technical Symposium on Computer Science Education, (pp. 267-271).

[28] Roessling G, Ackermann T. 2007. A Framework for Generating AV Content on-the-fly. Electronic Notes in Theoretical Computer Science :23-31.

[29] Roessling, G., & Freisleben, B. (2000). Approaches for Generating Animations for Lectures. Society for Information Technology & Teacher Education International Conference. San Diego, California.

[30] Roessling, G., & Freisleben, B. (2002). Animal: A System for Supporting Multiple Roles in Algorithm Animation. Journal of Visual Languages and Computing, 341-354.

[31] Schuler, M., Freisleben, B., & Roessling, G. (n.d.). The ANIMAL Algorithm Animation Tool.

[32] Seppala O, Karavirta V. 2009. Work in Progress: Automatic Generation of Algorithm Animations for Lecture Slides. Electronic Notes in Theoretical Computer Science :97-103.

[33] Shaffer, C., Cooper, M., & Edwards, S. (2007). Algorithm Visualization: A Report on the State of the Field. SIGCSE 07.

[34] Siff, M. (n.d.). Notes on AVL Trees. Retrieved 8 31, 2011, from Michael Siff's Homepage: http://pages.cs.wisc.edu/~siff/CS367/Notses/AVLs

[35] Stasko, J. (1990). The Path-transition Paradigm: a Practical Methodology for Adding Animation to Program Interfaces. Journal of Visual Languages and Computing, 213-236.

[36] Stasko, J., & Turner, C. R. (n.d.). Tidy Animations of Tree Algorithms.

[37] Stasko, J., (1998) Software Visualization (J. Stasko, J. Domingue, M. H. Brown & B. A. Price, eds). MIT Press, Cambridge,MA.

[38] Stasko, J., (1998) Samba Algorithm Animation System. Available at http://www.cc.gatech.edu/gvu/softviz/algoanim/samba.html.

[39] Stasko, J.T. (1990) Tango: A Framework and System for Algorithm Animation. IEEE Computer 23, September 1990, pp.27-39.

[40] Tamassia, R., Cohen, R., Shubina, G., Brandes, U., Goodrich, T. M., Hudson, B., et al. (n.d.). An Overview of JDSL 2.0, The Data Structures Library in Java.

[41] Zeil, S. (n.d.). ALGAE-Algorithm Animation Engine. Retrieved 9 1, 2011, from Dr. Steven J. Zeil's Homepage: http://www.cs.odu.edu/~zeil/algae.html

[42] Zeller, A. (2001) Animating data structures in DDD. First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press, Joensuu, Finland, 69-78.