Visualizing Live Data Structures

Curt Hill
Computer Systems and Software Engineering
Valley City State University
Valley City, North Dakota 58072
Curt.Hill@vcsu.edu

Abstract

This paper describes a project to visualize algorithms in an unusual way. The typical algorithm visualization produces an animation that may or may not be modified by user-entered data. This project takes an existing C++ program and produces snapshots of the current state of a pointer-based dynamic data structure. It then produces a three-dimensional representation of this data structure in an X3D file. There are no unusual restrictions on the pointer-based data structure.

This approach requires that the data structure be augmented with calls from a library. These calls register the items to display and the pointers of each different type of node. After this is done and the data structure is built, a snapshot routine is called to display the current state of the dynamic data structure. As many such displays as desired may be produced during the run of the program. Each of which is produced in a separate file. If a debugger is used to pause the program, a file that has been produced may be displayed while the program is paused.

1 Introduction

The data structures course is usually sufficiently challenging that pedagogical aids are helpful, if not required. One of the common aids are algorithm visualizations. The abstract concept of a pointer is difficult to grasp for the average student, but when the data structure is drawn with data and arrows representing the pointer, then it becomes somewhat easier. This pursuit has been explored for some decades, for example, consider Naps et. al. [5], which is by no means the earliest.

The simplest variety of visualizations function much like a movie with very little user interaction. The more complicated ones allow the user to enter the values to be added or removed, allowing the user to have a better understanding of how the data structure operates. However, the form of the data structure is solely the animator's choice. A web search will show a large number of these, such as [2], [4] and [6]. The web sites range from commercial ventures to free products of faculty or universities. Some are simple animations, while others provide tools to create animations. Several of these provide code examples.

Algorithm visualizations and animations are a great help for understanding the concepts of data structures. Without intent to minimize their usefulness, they offer very little in looking at an actual working or non-working program's state. For this, the debugger provided with the compiler is often the most useful tool. There is nothing wrong with debuggers and most developers find them helpful, yet showing a large data structure is often a tedious and error prone process with a debugger. To make matters worse the debugger display has to be built each time the program runs.

Another possibility is to have the program itself build a representation of the data structure. This generally involves an iterator processing the data structure. This is feasible for a production program, but for students struggling to get the data structure right in the first place it is of very little help. Such an addition to their code is a level of complexity comparable with the difficulty of developing the data structure in which they are struggling in the first place.

This project demonstrates is a library of routines that generalizes the iterator function and produces a file displaying the data structure. These are used to augment an existing C++ program to produce the files. The rest of this paper elaborates on these statements. Section 2 considers the usage. Section 3 briefly describes some of the inner workings. The final section discusses future work.

2 Usage

This system is a library of classes that display the data structure. The only class that the user deals with is the Visual Data Structure interface class, which has a class name of VDS. It has five methods, four of which are involved in the registration process and that last of which generates the visualization. The VDS class is generally a part of the main

program and then passed to the data structure class as part of the registration and display methods. The next section considers more detail of the VDS class and it numerous support classes. Here the use by an arbitrary program and data structure is considered.

The common C++ code using pointer-based data structures generally uses one or more classes that contain significant private or protected data. This generally requires using modification of the code itself to produce the visualizations. It would be possible to use structs, but the lack of information hiding is not the preferred practice in a language like C++. However, this system should work with C data structures and program as well.

The code modification involves augmenting the data structure with two steps. The first is to register the class information and the second is to generate a snapshot of the current data structure. These usually require two additional public methods to one of the data structure classes. These methods have access to the private and protected data of class so they may reveal to the VDS class the items needing display.

2.1 Registration

VDS makes no assumptions concerning the number or form of each class to be displayed. Therefore, it requires a registration process, where code with access to the needed variables describes them to a VDS object.

VDS assumes that there are two basic kinds of data in an object that is referred to by a pointer. First there are displayable items, such as numbers and strings. Second there are pointers to other objects.

There are four registration methods of the VDS class. The registration process starts with a call of the VDS method start_registration and ends with an optional call of method end_registration. Unlike dynamic languages, a C++ object type is a compile-timer entity and does not change at run-time. There are also two add methods of VDS to register the two different kinds of member data, add_pointer and add_display.

The VDS library is compiled into a library file. It knows nothing of the names of the objects in question. In order to do its work, it generally converts pointers into a void * form and uses unsigned integers as lengths. It will only be able to display names if they are passed as strings. This process would be somewhat easier in a language like Java that stores more compile time data in the code.

The start_registration method takes a single parameter which is the length of the object to be registered. This is obtained in the caller by the sizeof operator. This parameter is used by both add methods to check that the item passed is within the size of the object. The method returns and unsigned integer that gives a registration number for that object. All other methods that reference that object must provide the registration number, so that the code knows to which object this refers. This is the first parameter of both add methods.

The add_display method takes five parameters. The first is the registration ID previously mentioned. It takes two pointers. The first refers to the beginning of the object and the

second to the displayable item in question. The difference between these two is the offset into the object for this item. The fourth item is an enumeration describing the type of the item to display. At the time of this writing only five enumerations were available: DTInteger, DTFloat, DTDouble, DTString and DTStringObject. The last parameter is a string, which is the compile-time name of that item. Only the items of interest in the display of the object need to be registered.

The add_pointer method is similar in construction. The registration ID must be passed, as well as the beginning address of the object and the address of the pointer. The compile-time name of the pointer is also passed. The enumeration of the displayable types is not needed, but instead the registration ID of the object to which the pointer refers is needed. There is a predefined constant SELFID which is the registration ID of an object that points at its own type of object.

In summary, the registration process starts with a call to start_registration. This is followed by as many calls to add_display and add_pointer as needed. An optional call to end_registration concludes the process. After this as many display files may be generated as needed. Some examples follow in Section 2.3.

2.2 Displaying the data structure

Once registration is complete the program may begin to display its pointer-based data structure. The registration may precede or follow the building of the data structure, but clearly its display must be after there is something to display.

The VDS method to display is named snapshot. It requires three parameters and has an optional fourth. A pointer to the data structure is required, as is the registration number of this type of pointer. This pointer is usually in the private or protected data of the class of interest, but there is no reason it could not be a stand-alone pointer. It is most often on the run-time stack rather than the heap. The last required parameter is the file name that the display should be written upon. The optional parameter is a Boolean indicating to write a text version of the file or not, which is typically a debugging feature.

Due to historical accident the output is a 3-D visualization in the form of an XML variant known as X3D[7]. This is an ISO standard for representing 3-D scenes.

The snapshot builds a transparent box for each object. Inside the box are text lines. The first such line is the address of the object. The subsequent lines are the display items and the pointers that are contained in this object. If the compile name was included in registration, then it is displayed as well. The box is sized to contain all of these text items. The boxes are displayed in memory address order, with the lowest memory item lowest in the display. These boxes form a broken column as is seen later in Figure 1. The object that was initially pointed at has an arrow pointing at it from the left. There will be arrows from pointers to objects, but these were not ready at the time of writing.

The snapshot method may be called multiple times, preferably with different file names. This allows a visualization at different stages of the data structures development.

The snapshot starts a recursive process that displays each object. It records each pointer that it examines so that it will not pursue a circular path of pointers. It also distinguishes between valid pointers, invalid pointers and null pointers.

A null pointer is just zero valued, so it is comparatively easy to check. However, a valid and invalid pointer are somewhat trickier to distinguish. During construction of a VDS object three pointers are obtained and then released. The first and last give the bounds of valid heap memory. The middle one is a block large enough to contain all the objects a normal student program might obtain. When a pointer is examined it is determined to be valid if the address falls in the predefined range and invalid otherwise. Although it is possible that an invalid pointer is in the correct range, it is unlikely. Even if the unlikely occurs, the damage that can be done is minimized.

2.3 Example code and display

The test data used for this project was simply old demonstrations given in various classes. This includes a straight singly linked list, a circular singly linked list and a binary tree. The straight singly linked list is one of the simplest of dynamic data structures, so is a good example to consider in detail.

2.3.1 Straight singly linked list

The header file for this list contain only an integer key, a string object and the pointer to the next object:

```
class LinkedNode{
  private:
   int key;
   X3D_String data;
   LinkedNode * next;
   LinkedNode();
   friend class LinkedList;
};
```

The X3D_String is a typedef for the string object of the system in question. The system uses conditional compilation so that it works properly on different C++ systems.

Notice that the node is all private, with LinkedList given as a friend. The LinkedList class has all the public methods. The header for the LinkedList class:

```
class LinkedList{
 private:
   LinkedNode * root;
 public:
   LinkedList();
   ~LinkedList();
   bool add(int,X3D_String);
   bool remove(int);
   X3D_String find(int)const;
```

```
   int count();
   unsigned register_class(VDS & v);
   void snapshot(VDS & v, unsigned id, char * filename, bool list=false);
};
```

The LinkedNode and LinkedList classes are notably ordinary, but the two odd methods of LinkedList need further consideration. The register_class is the more complicated of the two:

```
unsigned LinkedList::register_class(VDS & vds){
 LinkedNode ln;
 unsigned id = vds.start_registration(sizeof(LinkedNode));
 bool ptr = vds.add_pointer(id, &ln, &ln.next,"next");
 bool num = vds.add_display(id, &ln, &ln.key, DTInteger,"key");
 bool str = vds.add_display(id, &ln, &ln.data, DTStringObject,"data");
 return id;
}
```

This code registers one pointer and two display items. It fails to call the end_registration method, but the first call to snapshot will end registration as well.

The snapshot method is largely a pass-through method to the VDS object:

```
void LinkedList::snapshot(VDS & v, unsigned id, char * filename,
                          bool list){
  v.snapshot(root,id,filename,list);
}
```

This is a method of the LinkedList class, but it could also be called from the main program.

Finally, after adding four items to the list, Figure 1 shows the visualization. The viewer used is freeWRL[1], a GNU X3D viewer.

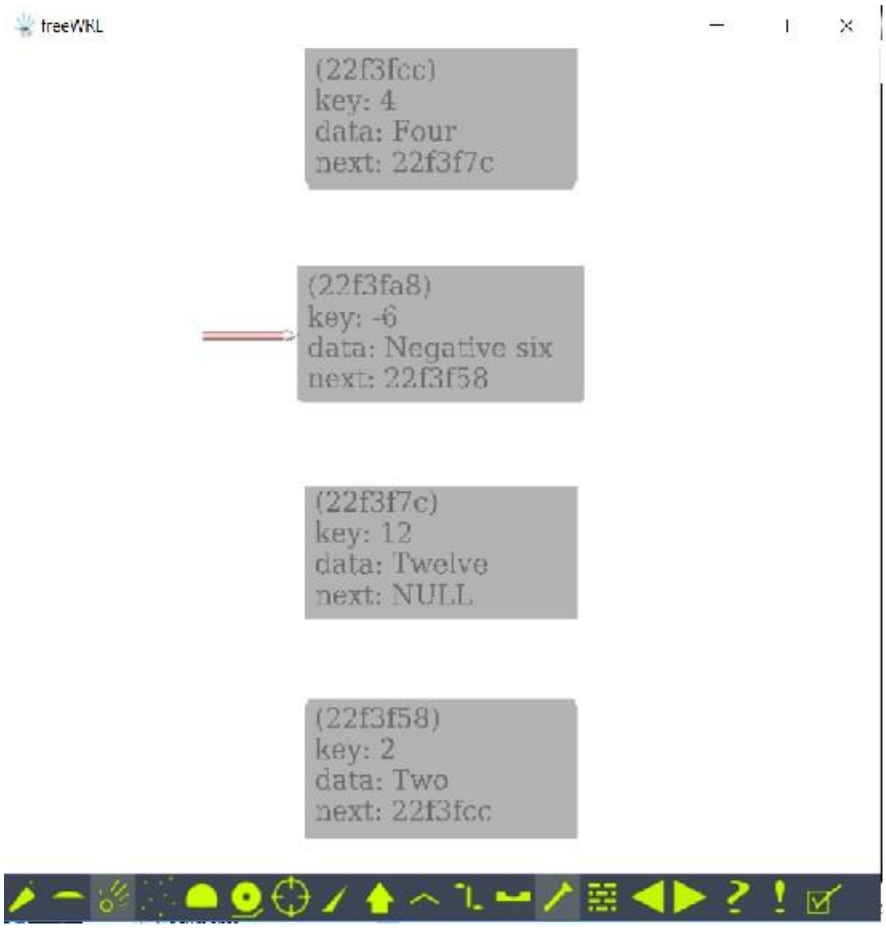

Figure 1 The Visualization of the List

Since the boxes are ordered by memory address rather than position in the list, the order reflects the order of entry into the list.

2.3.2 Circular singly linked list

A circular linked list, like an unrestricted graph, has cycles. VDS prepares for this by recording what pointers have been examined. In Figure 2 there are two displays. The leftmost is a good circularly linked list and on the right one where a pointer has been intentionally damaged.

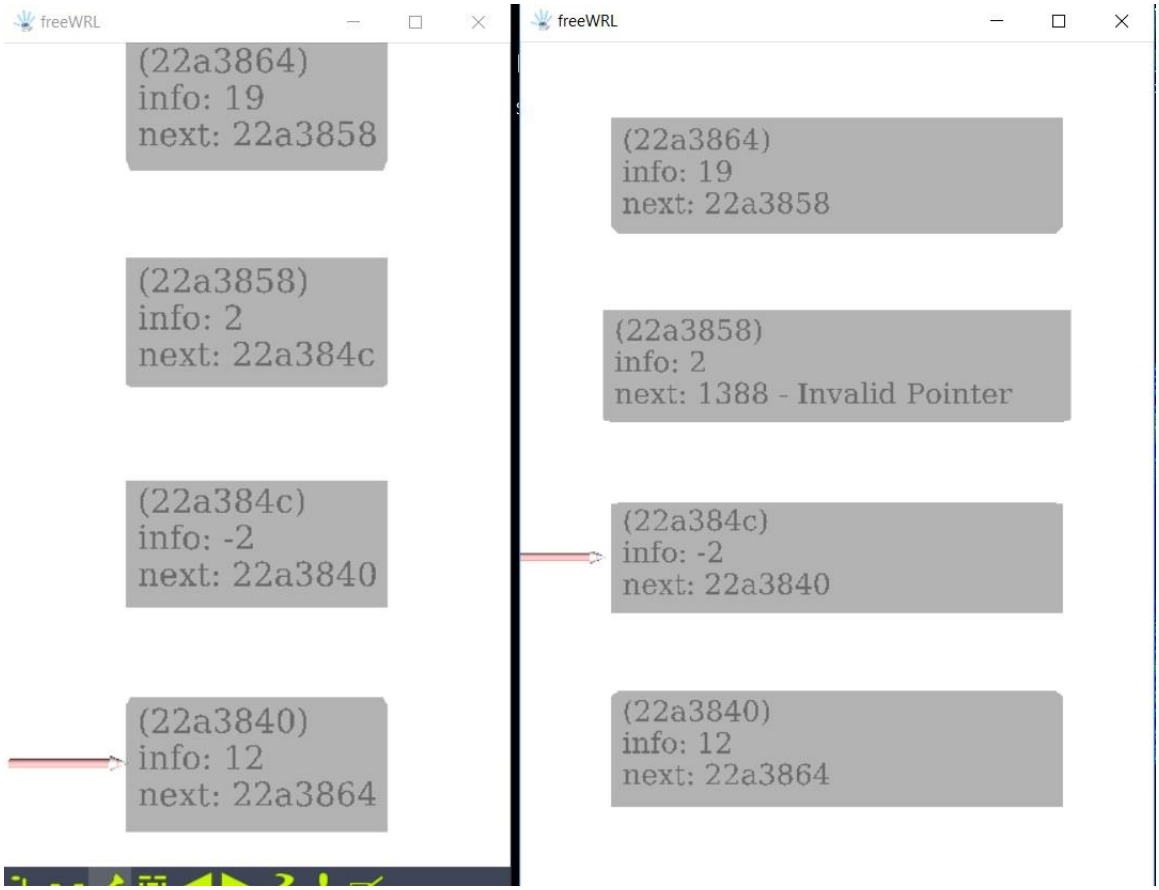

Figure 2. Two circular linked lists

The sizing of the boxes is different since the sizing is applied after the text is measured. The list on the right had its boxes stretched due to the “invalid pointer” message.

2.3.3 Binary Tree of Criminals

The example used in this case is a binary tree, where the subject matter is criminals and crimes. The data was generated by a Test Data Generator[3], so that the names are purely fictitious. Figure 3 has this display.

At the resolution of the paper, it is very difficult to be able to read the boxes. However, the viewer is much clearer, so the user has the ability to read multiple boxes.

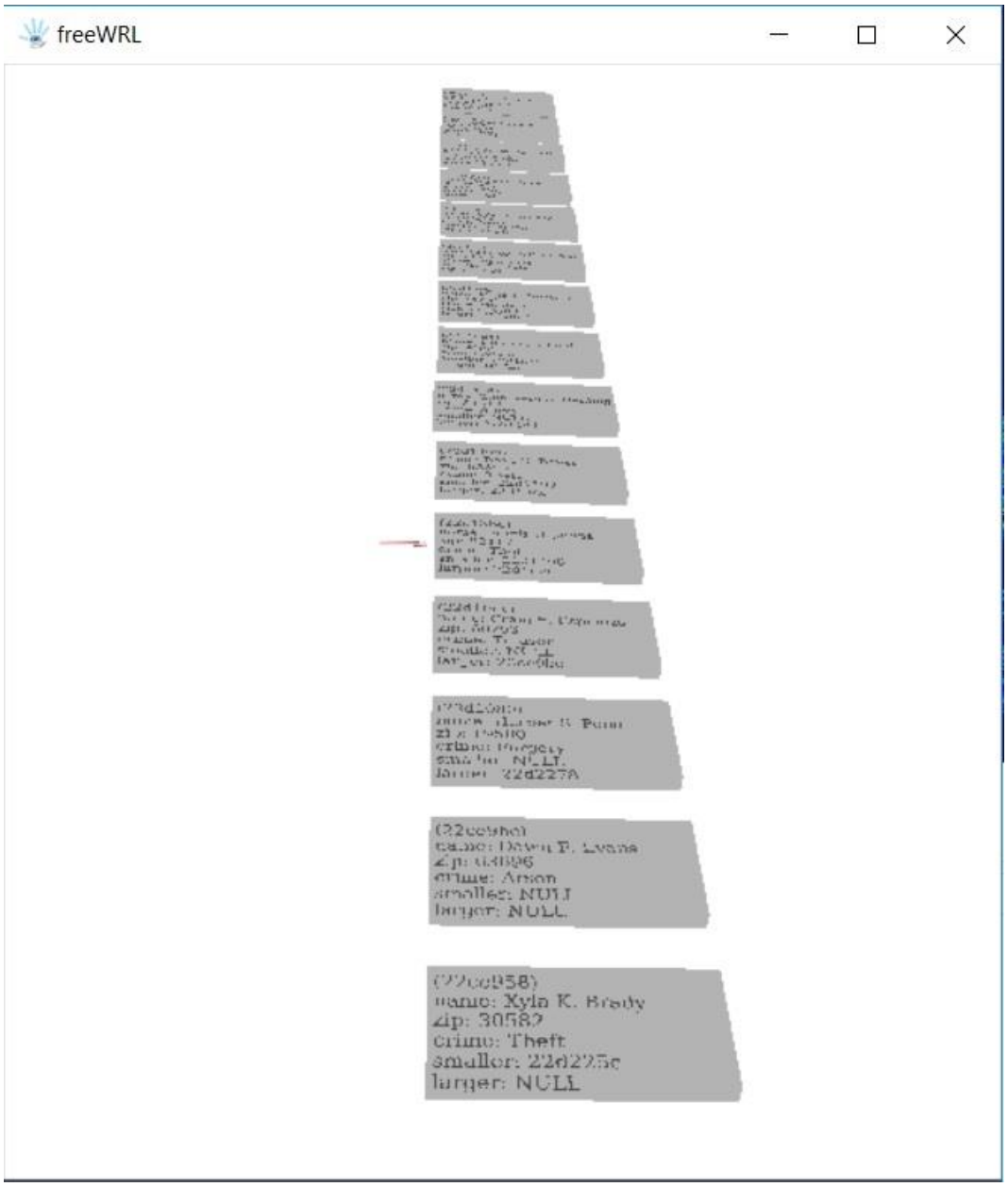

Figure 3 A tree with different perspective

The perspective has been altered here so that more boxes are visible.

This demonstrates several pertinent claims. Multiple pointers and multiple displayable items in an object are not a problem. Large numbers of objects in the dynamic data structure is also not a problem. Contrast this with a debugger display.

3 Brief Implementation Details

There are two layers of code in this project. The lower layer is a collection of classes that create an X3D file. The upper layer contains the support classes for the VDS classes.

The X3D generation layer contains several classes. The X3D class encapsulates the scene. There are two support classes, one representing a color and the other a 3D coordinate. Finally, there is an hierarchy of X3D solids, that represent basic shapes known in X3D.

The X3D_Solid class is the abstract base class of all of the solids. Descended from that are solids that X3D knows about: box, sphere, cylinder, text and cone. In addition to these is the group, which is not a solid, but a collection of solids that may be manipulated as one object.

The X3D classes could be used in a variety of other instances, even though they were designed to support visual data structures. Two additional shapes were added purely for the visual data structures project. The first is a block, which is a box with text inside it, which is the solid of a pointer based object. The second is an arrow, which is a cylinder and cone. Both of these are derivations of the group object.

The upper level of the system has VDS as its interface. These are the classes that support VDS as to display the pointer based objects. Most of these are not likely as reusable as the X3D classes.

4 Future work

It should be no surprise that there is still plenty of work to do with this project. There is a list of modifications that need to be started and few less that need to be completed. These include:

- The blocks currently give hexadecimal addresses as the representation of the pointer. The intent was always to fashion 3D arrows that would connect valid pointers with the destination block.

- The blocks currently form a column in memory order, which is a helpful view. The other helpful view would be to put the blocks in logical order. For a list this would still be a column, but for a tree it should likely have the more traditional triangular shape.

- The X3D representation level of classes was functional on two distinct compilers, a Borland C++ compiler and a Dev C++ compiler. This was enabled by a variety of preprocessor conditional compilation statements, since the two systems handle string object concatenation differently among other things. Adding the preprocessor statements for the Microsoft compiler has begun. These need to be finished for the whole system.

- Finally, and perhaps most importantly, the system has not been available to students or used by instructors to assist learning of the concepts and usage of pointer-based data structures. This will be the proof of the pudding.

It seems unlikely that this is a complete list.

In light of the fact that this system has not been used on the intended target, namely students learning data structures, no conclusions on the viability of this system may be stated.

Many languages more recent than C and C++ have moved away from the dangerous pointers in these languages. Most of these, such as Java, have attempted to tame the pointer or hide it. Yet the dynamic data structure still exists in many of these. The concepts of this system could still be applied in different languages to visualize live data structures.

References

[1] FreeWRL. FreeWRL Home Page. http://freewrl.sourceforge.net/ Date accessed March 2018.
[2] Galles, David. Data Structure Visualizations. https://www.cs.usfca.edu/~galles/visualization/Algorithms.html. Date accessed February 2018.
[3] Hill, Curt. Instructor Tools: Test Data Generation. MICS 2010. April 16-17 University of Wisconsin – Eau Claire.
[4] Meech, Duncan. Algomation. http://www.algomation.com/. Date accessed February 2018.
[5] Naps, Thomas L., James R. Eagan, Laura L. Norton. JHAVÉ – An Environment to Actively Engage Students in Web-based Algorithm Visualizations. Proceedings of SIGCSE 2000. Pages 109-113, Austin, TX. March 2000.
[6] VisualGo.Net. *Visualizing data structures and algorithms through Animation.* https://visualgo.net/en Date accessed February 2018.
[7] Web3D Consortium. What is X3D? http://www.web3d.org/x3d/what-x3d. Date accessed March 2018.