Algorithm Explorer: Visualizing Algorithms in a 3D Multimedia Environment

[Extended Abstract]

Erik Carson
Department of Computer Science & Engineering
University of North Texas
Denton, TX, USA
carson@cs.unt.edu

Ian Parberry
Department of Computer Science & Engineering
University of North Texas
Denton, TX, USA
ian@unt.edu

Bradley Jensen
Department of Information Technology & Decision Sciences
University of North Texas
Denton, TX, USA
jensenb@unt.edu

ABSTRACT

Computer science courses have increasingly made use of visualization tools to illustrate common algorithms. This paper describes Algorithm Explorer, an educational tool designed for use by instructors and students to examine algorithms in a rich environment composed of 3D data representations, 3D audio cues, and easy-to-use controls. Instructors and students can easily add calls to Algorithm Explorer's C++ API to their programs to quickly develop engaging visualizations, and every detail of the scene can be customized as the developer desires.

Categories and Subject Descriptors

K.3.2 [**Computing Mileux**]: Computers and Education-Computer and Information Science Education[Computer science Education]

General Terms

Design, Experimentation

Keywords

Algorithm animation, visualization

1. INTRODUCTION

Algorithm Explorer is an educational tool designed to help computer science students visualize many common algorithms used in a standard computer science curriculum. The tool consists of a Microsoft Windows application for viewing and manipulating visualizations, a C++ API for developing visualizations in Microsoft Visual Studio, and an add-in for Microsoft Visual Studio that supplements the API and simplifies the development process. The visualizations themselves are rich multimedia presentations implemented using DirectX 9.0 and feature three-dimensional representations of common data structures, animation of common operations, context-sensitive tracing of source code or pseudocode, and support for sound effects that reinforce the animation. Algorithm Explorer is highly customizable on both the user and developer levels. Developers can set parameters for the colors, textures, lighting, and sound effects for various structures, operations, and events. Users can view these scenes as developed or override these with their own customizations at runtime.

Instructors can use Algorithm Explorer as a primary tool for presenting algorithms or as a supplement to textbooks, slide shows, and other traditional forms of presentation. Students can use Algorithm Explorer's graphical interface to view visualizations provided by their instructor. They can also use the C++ API as a comprehensive aid and a debugging tool when developing or working with C++ implementations of common algorithms.

The main part of this paper is divided into three sections. Section 2 describes prior work and compares and contrasts it with Algorithm Explorer. Section 3 describes Algorithm Explorer in more detail from the user and programmer perspectives. Section 4 contains two examples of algorithm visualization code using Algorithm Explorer.

SIGCSE'07, March 7–10, 2007, Covington, Kentucky, USA.

2. PRIOR WORK

Several systems already exist for visualizing algorithms, each with its own philosophy and range of application. This section briefly examines several systems and reflects on their merits, drawbacks, and influences on Algorithm Explorer.

2.1 Tango, XTango, Polka, and Samba

Stasko developed the Tango framework for algorithm animation in 1990 [8]. In XTango (the X11 Window implementation of Tango), animations consist of a control file that contains graphics instructions grouped into events that correspond to key parts of the algorithm [9]. The developer interleaves XTango API calls into a C implementation of the algorithm. With enough effort, an animation designer can create almost any imaginable 2D or 2.5D representation using primitive drawing and animation operations. How-

ever, such animations require explicit placement and low-level setup of every component of the animation. Stasko later developed Polka to visualize parallel algorithms and Samba to allow designers to build animations via text scripts [10, 11].

2.2 Animal

In Rößling's Animal system, animations are built using a graphical editor [7]. The interface generates a text script which can then be modified by hand. Like Tango, Animal can be used to visualize many representations; however, like Tango, it does so through the composition of low-level drawing and animation operations rather than operations on the abstract data types represented in the animation; in fact, Animal animations can be generated without actually implementing the algorithm in code. Animal has specialized support for displaying code or pseudocode in the animation. Finally, Animal has a sizable online repository of animations that can be retrieved through the interface itself [6].

2.3 MatrixPro

The MatrixPro system by Karavirta et al. takes another approach to algorithm visualization [1]. The user builds animations in a graphical interface by choosing from a menu of data structures. The user can then use drag-and-drop operations to manipulate the data structures interactively. The data structures update themselves according to their expected behavior. All operations on the scene are recorded and can be saved to a file for later review. Furthermore, operations can be layered at different levels of granularity and played back at any given level, skipping finer steps as desired. MatrixPro's primary advantage is that its users can alter the visualization without the need to recompile or reload an animation. However, MatrixPro has no support for source tracing, and creating new data structures is complicated and requires extensive Java coding on the part of the designer [5].

2.4 CS Teaching Aids for Visual J# .NET

Microsoft has released visualizations of sorting and recursive algorithms using Visual J# .NET [4]. Visual Studio .NET is required to build these teaching aids.

The sorting visualization compares two of three basic algorithms for sorting arrays—insertion sort, merge sort, and selection sort—in parallel using a bar graph representation of the arrays. The algorithms are compared based on elapsed time and number of comparisons. While the visualization includes a description of each algorithm along with source code, the visualization does not trace the source code as the program progresses. The recursion visualization demonstrates several recursive functions by stepping through each function call and return. The steps are visualized as boxes on the right side of the window. These boxes contain either the function's sole parameter or its return value. Several common functions (e.g., factorial, Fibonacci) are built into the program. Both programs can be extended to visualize other algorithms (within some constraints).

2.5 Jeliot 2000

Jeliot 2000 is a Java programming environment that automatically animates a subset of Java source code [3]. Jeliot 2000 animates Java code on a very low level, displaying every Java instruction (e.g., comparison, addition) and creating stack frames for method calls. Arrays are visualized in their own panel, and array indices are visualized as arrows pointing to the appropriate element. Jeliot 2000 seems more suited to an introductory programming course; an undergraduate algorithms course usually views data structures on a more abstract level. Jeliot 2000 works with most of Java's language elements, but it cannot visualize objects other than strings, making it unsuitable for viewing, for example, high level graph operations. Jeliot provides few options for customizing the animation apart from play speed and fonts.

2.6 BlueJ

BlueJ is a visual programming environment designed for introductory courses in object-oriented programming using Java [2]. While not specifically an algorithm visualization system, BlueJ does visually represent all of the classes and instances that make up a Java application, including Java API classes. Rather than developing linear programs that run from a static main function, users of BlueJ write functions that can be called at will from the interface. Like Jeliot, the visualization is too low-level to visualize abstract data types such as graphs and trees without considerable coding. However, the system is excellent for visualizing objects at the class level.

3. DESCRIPTION

Algorithm Explorer addresses a number of problems with previous efforts at visualization:

- Developers can create visualizations for algorithms in less time than they spent implementing the algorithms themselves, though further effort can refine a visualization's audio-visual aspects.
- Visualizations are high-level, on par with the data structure itself (e.g., heap) or the underlying implementation (e.g., array used to implement a heap) as desired.
- Any algorithm that can be represented by a combination of blocks, arrays, call stacks, and graphs can be visualized with Algorithm Explorer.
- Visualizations support source tracing, and the add-in for Visual Studio .NET 2005 automates much of the source-tracing process.
- The visualizations are very media-rich. 3D graphics, 3D audio, and smooth animation all combine to make a more immersive visualization experience for the learner.

Algorithm Explorer is used on two levels—by the user interacting with the visualizations and by the programmer creating visualizations. The user sets different visualization parameters using the GUI controls and plays visualizations. The programmer uses Algorithm Explorer's C++ API calls to construct and configure visualizations. The user interface is built on top of the programmer interface, so the programmer can have as much or as little control over the user interface as desired.

3.1 User Interface

The visual interface of Algorithm Explorer consists of a top-level application window with the following child components:

- a viewer panel that displays the currently loaded visualization scene at the current time

- a play control panel that controls the progression of the visualization in a manner analogous to the controls of a compact disc player
- a speed panel that controls the speed of the visualization
- an option panel containing controls for altering various parameters of the visualization such as lighting, projection, sound, and rendering states
- a menu bar containing the commands for working with visualization files, viewing the help files, and showing or hiding the various panels

The user interface can be in one of two visual modes. In standard mode, all of the components are visible. In fullscreen mode, all components except the viewer panel are hidden, and the viewer panel is expanded to fill the entire screen. The user can switch between standard and fullscreen modes with a shortcut key.

The primary device for interacting with these controls is the mouse. However, the user can also use keyboard commands to adjust the controls in a manner similar to that used in other Windows applications (e.g., pressing ALT and an alphabetic key to select a control).

3.1.1 Viewer Panel

The viewer panel is a viewport into a Direct3D scene displaying the currently loaded visualization at the current time. During playback, the graphical portion of the visualization will be animated here. In standard mode, the user can resize the viewer panel by resizing the window; the viewer panel expands to fill the extra space. In fullscreen mode, this panel fills the entire screen. The user can use the mouse to navigate the scene in arc-ball fashion, zoom in and out, and reset to the default view.

3.1.2 Play Control Panel

The play control panel allows the user to visualize different moments in visualization time. The panel consists of a row of play control buttons, a time scrollbar, and a step scrollbar.

The buttons afford playing, pausing, and traversing the visualization similarly to the buttons on a conventional compact disc or digital video disc player. Using these buttons, the user can play, stop, pause, and resume the visualization; move backward or forward by a step; or jump to the beginning or end of the visualization.

The time and step scrollbars indicate and control the current moment in visualization time. By sliding one of the scrollbars' thumbs, the user can navigate the visualization in clock time or steps. Text labels beside each scrollbar indicate the current step and moment in time.

3.1.3 Speed Panel

The speed panel consists of a slider allowing the user to adjust the speed factor of the visualization. By default, the speed factor is 1.0, indicating that all times specified in the visualization are exact; for example, if the visualization indicates a three second delay between steps, then the visualization will pause for three seconds between steps when playing. Sliding the speed slider's thumb to the left or right decreases or increases the speed factor respectively, thus slowing down or speeding up playback. A text label in the panel displays the current speed factor, and a button resets the speed factor to the default value.

3.1.4 Option Panel

The option panel contains a number of controls for adjusting various parameters for visualizing the scene. For example, the user can clear or check boxes that enable lighting, wireframe rendering, or orthogonal projection; or the user can use a color picker to change the color of all highlighted edges in a graph.

3.1.5 Main Menu

The main menu bar contains commands for opening and closing files, viewing the Algorithm Explorer documentation files, and modifying the interface itself by showing and hiding various panels. Additionally, any other useful commands that don't logically fall into the above panels are included in the main menu.

3.2 Scene Development Interface

The scene development interface consists of two parts: a C++ Application Programming Interface (API) and a helper add-in for Visual Studio .NET 2005.

3.2.1 C++ API

The C++ API contains all of the functionality needed to create scenes for use with Algorithm Explorer. Thus, existing programs used to demonstrate algorithms can be easily modified to produce visualizations.

The `Explorer` interface class provides access to the user interface and (if any) the currently loaded visualization. Visualizations can be developed independently of the user interface and saved to a file, or they can be generated on-the-fly and passed to the `Explorer` interface for immediate visualization. Saved visualizations can be retrieved either by the user application or the API.

The developer can manipulate the visualization itself via a `Scene` interface and a number of interfaces to the scene's components (e.g., data blocks, arrays, source code listings). These interfaces are organized into core components and components of three different scene types:

- *Block World* is used to visualize operations on individual data blocks or arrays represented by 3D boxes. The boxes can be configured to either contain a 3D text mesh representing their value or scale along one or more dimensions in proportion to their value.

- *Graph World* visualizes operations on graphs, which are represented by spherical nodes, curved edges, and 3D text representing the names and/or values attached to these components.

- *Stack World* illustrates recursion and general function tracing through a dynamically resizing stack of containers, each with its own set of components.

Of course, these types of scene can overlap; a single visualization can have elements from all three worlds at the same time.

The API operations used to generate steps in a scene typically either (a) update the data values of components to match changed data or (b) mirror the original data operations used to implement the algorithm. For instance, incrementing a data block with the `++` operator generates a step that visualizes a variable incrementing by one. A number of common procedures are also implemented for con-

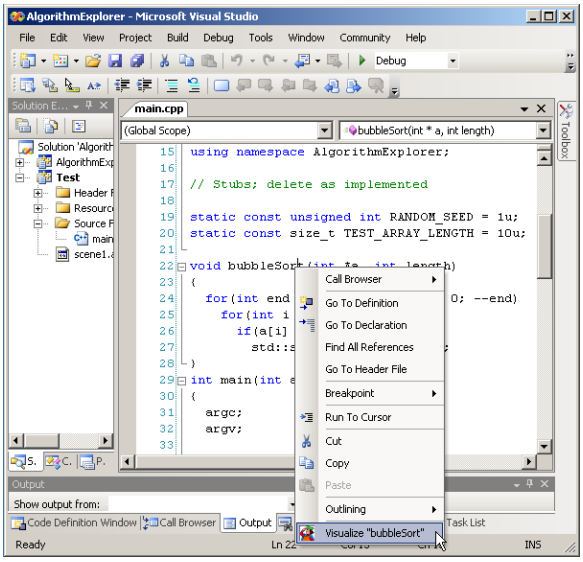

Figure 1: Visualizing a function with the add-in

```
void bubbleSort(int *a, int len)
{
  for (int j = len-1; j >= 0; --j)
    for(int i = 0; i < j; ++i)
      if(a[i] > a[i+1])
        std::swap<int>(a[i],a[i+1]);
}
```

Figure 2: A bubble sort function

venience, such as swapping the contents of two data components. These operations are fully animated, expressed through the movement of components or their values, sound effects, change in lighting, or any combination of the above. While default behaviors are provided so that the scene is highly immersive without much effort from the developer, the developer is free to modify any behaviors as desired.

3.2.2 *Visual Studio add-in*

To further aid in scene development, Algorithm Explorer comes with a helpful add-in for Visual Studio. The add-in can automatically add code to a C++ project that initializes Algorithm Explorer and sets up an empty scene. The add-in can also implement some of the scene building code for selected functions automatically. By choosing "Visualize" from the Algorithm Explorer context menu on a function header, the add-in will create a copy of the function with a modified name and an additional `SceneCreator` parameter representing the scene to be constructed. In addition, if source tracing is enabled in the add-in options, the add-in creates a third function that adds a visual component representing the original function's source code to the scene. The developer can manipulate this component using the `SourceBlock` interface.

4. EXAMPLES

Figure 2 shows a typical bubble sort implementation, and Figure 3 shows the same implementation with scene-building code. Some of the code, such as the extra parameter for the

```
void bubbleSort_ax(int *a, int len,
    SceneCreator scene)
{
  BlockArray a_ax =
      scene.getBlockArray("a", a, len);
  Block j_ax = scene.getBlock("j");
  Block i_ax = scene.getBlock("i");
  scene.beginScene();
  int j;
  for (j = len-1, j_ax = j; j >= 0;
      --j, j_ax = j)
  {
    int i;
    for(i = 0, i_ax = i; i < j;
        ++i, i_ax = i)
    {
      a.highlight(i,j);
      if(a[i] > a[i+1])
      {
        std::swap<int>(a[i],a[i+1]);
        a_ax.swap(i,i+1);
      }
    }
  }
}
```

Figure 3: Generating a visualization for bubble sort

`SceneCreator` interface, can be automatically generated by the Visual Studio add-in. The rest of the code consists of calls that create visual representations of the data involved followed by operations that update these representations.

Figure 5 constructs a visualization of a standard algorithm used to extract the lowest value from a min-heap. The code assumes the existence of a class named `MinHeap` which implements the heap as a member array `elems` of $2^k - 1$ elements, where k is the maximum number of levels in the heap; a counter `size` tracking the number of elements in the heap; and visualizations for these data elements. Additionally, `MinHeap` contains a member variable `stack` representing a function call stack. While this function is iterative, a stack provides built-in support for visualizing return values after a function ends. In addition, a larger scene could call this function (and others) many times, so a stack component keeps the visualization area tidy.

An interesting feature of this implementation is that the visualization can be that of the array-based implementation (as in Block World) or the heap itself (as in Graph World) depending on the interface used to define the member variable `elems_ax` in `MinHeap`. If `elems_ax` is a `BlockArray` interface, the visualization is that of an array of blocks containing the heap's elements. However, if `elems_ax` is a `BinaryHeap` interface, the visualization is that of a balanced binary tree. Both interfaces can be manipulated by using the subscript operator `[]` to indicate the node index. Thus, the scene developer can switch from low-level to high-level visualization by merely replacing the type of `elems_ax` and uncommenting a line.

5. CONCLUSION

Compared to prior algorithm animation systems, Algorithm Explorer provides richer visualizations that take better advantage of modern 3-D graphics and audio technology.

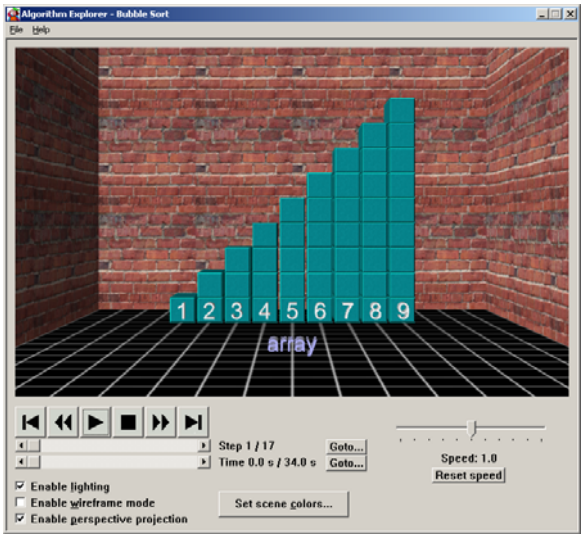

Figure 4: A bubble sort visualization

These visualizations are designed to improve students' learning experience by creating a more engaging and immersive environment. At the same time, the development API, with the aid of the Visual Studio add-in, is designed to fit neatly into existing algorithm projects with minimal training and effort on the part of the visualization designer.

This research was funded by a grant from Microsoft Corporation.

6. REFERENCES

[1] V. Karavirta, A. Korhonen, L. Malmi, and Stålnacke. MatrixPro: A tool for on-the-fly demonstration of data structures and algorithms. In *Proceedings of the Third Program Visualization Workshop*, pages 26–33, 2004.

[2] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education*, 13(4):249–268, 2003.

[3] R. B.-B. Levy, M. Ben-Ari, and P. A. Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 40(1):15–21, 2003.

[4] Microsoft Corporation. Computer science teaching aids for Visual J# .NET. `http://msdn.microsoft.com/vjsharp/using/academic/teaching/default.aspx`, Accessed November 9, 2005.

[5] S. Pollack and M. Ben-Ari. Selecting a visualization system. In *Proceedings of the Third Program Visualization Workshop*, pages 134–140, 2004.

[6] G. Rößling. Overview of the Animation Repository. `http://www.animal.ahrgr.de/animations.php3`, Accessed November 9, 2005.

[7] G. Rößling and B. Freisleben. Animal: A System for Supporting Multiple Roles in Algorithm Animation. *Journal of Visual Languages and Computing*, 13(3):341–354, 2002.

[8] J. Stasko. TANGO: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, 1990.

```
class MinHeap:
{
  public:
    // MinHeap implementation goes here,
    // including elems

    BinaryHeap elems_ax;
};

int MinHeap::extractMin()
{
  StackFrame frame =
      stack.push("extractMin()");
  int rv = elems[0];
  Block rv_ax = frame.getBlock("rv");
  rv_ax = rv;
  --size; size_ax = size;

  if(size > 0)
  {
    elems[0] = elems[size];
    elems_ax[0] = elems[0];
    unsigned int c;  // child
    Block c_ax = frame.getBlock("c");
    Block i_ax = frame.getBlock("i");
    for(unsigned int i = 0, i_ax = i;
        i*2+1 < size;
        i = c, i_ax = i)
    {
      c = i*2+1;  c_ax = child; // left
      if(elems[c] > elems[c + 1])
      {
        ++c;  c_ax = c; // right
      }
      if(elems[i] > elems[c])
      {
       // keep reheaping
        std::swap(elems[i], elems[c]);
        elems_ax.swap(i, c);
      }
      else
        break; // heap restored
    }
  }

  // Uncomment if using graph visualization
  // elems_ax.resize(size);

  return stack.pop(rv);
}
```

Figure 5: Generating a visualization for min-heap extraction

[9] J. Stasko. Animating algorithms with XTANGO. *SIGACT News*, 23(2):67–71, 1992.

[10] J. Stasko. Using student-built algorithm animations as learning aids. In *SIGCSE '97: Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, pages 25–29, New York, NY, USA, 1997. ACM Press.

[11] J. Stasko and E. Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, 1993.