Animation, Visualization, and Interaction in CS 1 Assignments

Owen Astrachan * and Susan H. Rodger †
Duke University
ola@cs.duke.edu, rodger@cs.duke.edu

Abstract

Programs that use animations or visualizations attract student interest and offer feedback that can enhance different learning styles as students work to master programming and problem solving. In this paper we report on several CS 1 assignments we have used successfully at Duke University to introduce or reinforce control constructs, elementary data structures, and object-based programming. All the assignments involve either animations by which we mean graphical displays that evolve over time, or visualizations which include static display of graphical images. The animations do not require extensive programming by students since students use classes and code that we provide to hide much of the complexity that drives the animations. In addition to generating enthusiasm, we believe the animations assist with mastering the debugging process.

1 Introduction

In this paper we describe a collection of CS 1 assignments that visualize and/or animate problems that are too complex or tedious to view in a textual manner. Each problem focuses on either objects changing over time, or a large amount of data to display. For those problems with objects changing over time, displaying their changes in a textual manner results in a large amount of output. This output is easy to decipher if just one object is moving, but tedious if multiple objects are moving at the same time, since the output descriptions are interlaced. For those problems with a large amount of data to display, we display the data graphically. In some cases, the order the data is generated is important, so its drawing is animated to provide a historical view.

Our assignments are written in C++ and most were created in 1996. Animations use either Xtango [8], its successor Samba [9], or a Java compatible version of these called Lambada [2]. Image display uses the shareware program XV, running under X Windows. We provide classes and code to hide the details of Xtango and XV so students can focus on using C++ while still able to generate animations and pictures. The Xtango animator is an interpretor that parses textual commands for creating and moving objects. Programs can be written in any language to print Xtango commands which are piped to the animator. The XV program reads a file as input. Code provided to students writes a temporary file, calls XV to display the file, and then deletes the temporary file making display seamless from a user's perspective.

In the next sections we give several examples of how we use animations and interactive graphics at Duke University to introduce language features and topics from computer science. In Section 2 we describe a basic use of animation we use early in the semester to introduce students to classes; in Section 3 we describe Xtango and a simple drawing program students write; in Section 4 we discuss animations based on random walks and a cardioverter/debrillator; in Section 5 we discuss an animated histogram program; and in Section 6 we discuss an interactive image processing program. Concluding remarks are found in Section 7.

2 Balloon Race

In this section we describe three assignments, one that introduces a first C++ class, a second assignment to create multiple objects of this class, and a third assignment that reinforces use of this class, introduces a second class and introduces looping constructs.

Students learn about classes early on in our CS 1 course at Duke. The first C++ class (other than a string class) we use simulates a hot-air balloon. A balloon can ascend, cruise, and descend. In a first assignment, students use the balloon class [1] to manipulate one balloon. Output describing the movements of the balloon every few meters is automatically generated in text format. Additional distractions such as wind shear make the balloon's behavior more realistic. For example, output may appear indicating the balloon suddenly dropped 5 meters due to wind shear.

In a second assignment students create multiple balloons and interlace their movements. Since the text output of multiple balloons is tedious to decipher, we incorporated Xtango commands into the balloon class to automatically generate animations [7] of the balloons. Students are aware that their

*This work is supported in part by the National Science Foundation Grant #DUE-9554910

†This work is supported in part by the National Science Foundation Grant #DUE-9555084

use of the public member functions is the same, and that the private member functions have been modified to produce animated output instead of textual output.

In a third assignment, students write an animated balloon racing program that uses a further modified balloon class, looping constructs for the first time, and a second class, the Dice class, for generating random numbers. The additional member functions for the balloon class are GetDistance to return the distance a balloon had cruised, and Flash to repeatedly flash (display on and off) a balloon. We provide the following guidelines to students.

1. Create three balloons that rise to 40, 60 and 80 meters.

2. Repeat 25 times: Cruise each balloon randomly 1-4 meters.

3. Determine which balloon cruised furthest (if any) and flash the winner moving the losing balloons to the ground.

The animation created is very simple. Each balloon is represented by a circle of a different color that moves as instructed. A snapshot of the Xtango animation with three balloons is shown below.

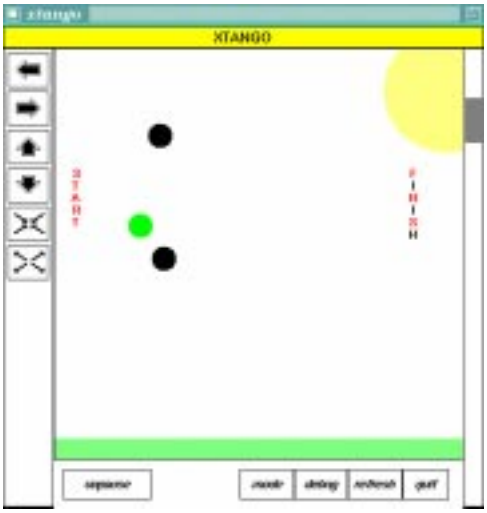


Figure 1: Snapshot of Balloon Race Animation

3 Graphics

In this section we give a brief description of animation commands used in Xtango (the same commands work in Samba or Lambada). Although the animation commands are simple, we use a class *Window* to encapsulate the commands so that students make calls of *Window* member functions rather than generating Xtango commands. Rather than learning both C++ syntax and Xtango syntax, the encapsulation enables students to concentrate more on the problem being solved than on a new language.[1] For example, to create a solid, 25×30, green-filled rectangle with lower-left coordinate at (12,40) the command below is used in Xtango.

```
rectangle 101 12.0 40.0 25.0 30.0 green fill
```

The 101 is a unique tag identifier used to move or hide the rectangle with subsequent commands. Use of the *Window* class is shown below where `Picture` is a *Window* variable.

```
int rectag;
rectag = Picture.Rectangle(12.0,40.0,25.0,30.0);
Picture.Color(rectag,"green");
Picture.Fill(rectag);
```

Rather than requiring students to keep track of Xtango tag identifiers, the tag is generated by the *Window* class and can be stored in mnemonically named variables. Other *Window* member functions include figure drawing functions:`Triangle()`, `Circle()`, `Line()`,; figure display functions: `Color()`, `Fill()`, `Outline()`, `HalfFill()`, `Toggle()`; and functions to construct and resize the drawing window. With this approach, student mistakes are more likely to trigger a C++ compile error than an Xtango error message that students have more trouble debugging.

3.1 A Simple Drawing Program

This CS 1 assignment introduces file streams and reinforces conditional and looping constructs using the *Window* class. Students write a program to read a data file containing simple commands describing a picture and convert the commands to calls of *Window* member functions to create the picture. Students implement the assignment in three steps using iterative enhancement to increase the functionality of an already working program:

1. Process commands to create rectangles, circles, triangle, and lines.

2. Process additional commands to color, fill and half-fill the most recently drawn object.

3. Ignore multiline comments in the data file and generate text in the picture.

For extra credit, students create and parse a new command to draw a rotated rectangle (by default rectangles are drawn with horizontal and vertical edges). The rotated rectangle has to be calculated and drawn as four lines since there is no *Window* member function for this. Students create and submit the drawings they produce; one of the student drawings is reproduced in Figure 2.

[1] The class also permits symbolic identifiers for Xtango objects, a feature now incorporated in Samba and Lambada but not available when we first developed the assignment.

Figure 2: Output of Drawing Program

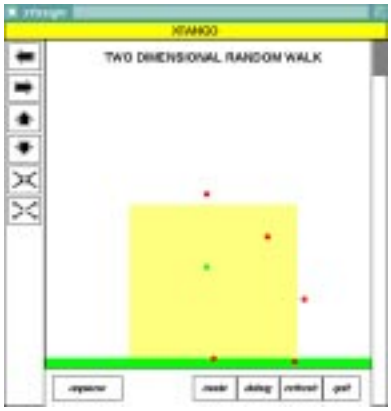


Figure 3: Snapshot of Two Dimensional Random Walk

4 Random Walks and Heart Monitors

A random walk is a model built on mathematical and physical concepts used to explain (among other things) how molecules move in an enclosed space. In our assignments we use both one- and two-dimensional random walks. In this section we discuss an animation based on a two-dimensional walk. This assignment builds on the one-dimensional random walk by modifying an existing class, introduces pass-by-reference parameters, revisits the Dice class, and reinforces looping constructs.

We first discuss the classes used to simulate a one-dimensional random walk. We use two classes: a *RandomWalk* class that simulates a frog jumping left and right at random, and a *WalkObserver* class that observes the random walk. The observer class is an example of a design pattern [4] that appears again in the simulation of a cardioverter/debrillator discussed in Section 4.1.

We supply an observer class that tracks how far left and right one frog travels during a simulated walk. We ask students to modify the observer class to use the *Window* class described in Section 3 that drives the Xtango/Samba animator. Students then modify the simulation to track several frogs that hop two-dimensionally: north and south in addition to east and west. Frogs always begin on a square dock that is surrounded on three sides by water and on one side by grass. We limit the dimensions of the dock the frogs inhabit so that reaching the edge causes a frog to jump off the dock. This requires modifying the walk class and the observer class to leave a mark where the frog jumps off the dock. A snapshot of this simulation is shown below for six frogs (shown as circles); four have jumped off the dock (two into the water and two into the grass).

Later in the course we return to this simulation and ask students to use an array to track all locations of a one-dimensional random walk. This could naturally lead to a visual histogram similar to the one described in Section 5.

4.1 Monitoring a Heart

A cardioverter/debrillator monitors heart beats and shocks the heart if it is beating too rapidly or too slowly. This assignment is based on work described in [5] although extended and modified for an object-oriented and graphical approach as described in this section. The focus of this assignment is the modification of a class through iterative enhancement.

The simulated heart-monitor illustrates a key benefit of simulations: modeling the real world before coping with the real world. This assignment is difficult enough that students appreciate being able to debug a simulation rather than a real pacemaker.

This program uses two classes: a simulated heart, and a heart monitor or observer. The framework is similar to the random walk thus reinforcing the design pattern. Initially the simulated heart reads data from a file. However, we also ask students to simulate randomly-fluctuating sinusoidal rhythms in which both the period and amplitude change to simulate different heart rates. The parameters of the sine function are stored as data members so that when the heart is "shocked" the parameters can be changed thus literally changing the way the heart beats. We first used a text-based observer, but then incorporated an Xtango animation that simulates an oscilloscope. For extra credit students can simulate rarely occurring flatlines and rescitations using simulated shocks. A static view of the simulation is shown in Figure 4. On the left side there is a vertical line indicating the heart was shocked.

5 Animated Histogram

This assignment reinforces the use of arrays and introduces an array of structs via the inventory of a fish store represented as a histogram. The name of each item is displayed and a series of tick marks representing the quantity of the item is shown horizontally.

We use the *Window* class described in Section 3 to draw the picture. The `Text()` function draws the name of each item in a column on the left hand side of the window. The

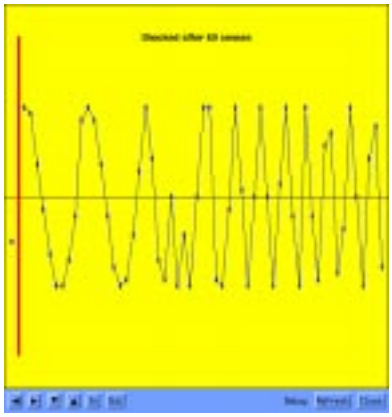


Figure 4: Snapshot of simulated heart beat

`Toggle()` function is used to hide or redisplay a tick mark as an item is bought or sold. The unique tag of tick marks must be saved in order to later hide the mark. For each item in the store, an array is used to store the unique tags of its tick marks.

The assignment has two parts and extra credit. In part 1, the items and their initial quantity are read from a data file and displayed. Tick marks are shown in black indicating the items they represent are in stock. In part 2, additional information representing sales and purchases and their quantity are read in from the data file. When items are sold, the corresponding black tick marks are removed. Sometimes stores sell items they do not have in stock, but they know are coming in soon. Items on backorder are represented by red tick marks. When items are bought for resale, additional black tick marks appear or red tick marks may disappear if the items were on backorder. For extra credit, the inventory is displayed in alphabetical order. Figure 5 shows a snapshot of the histogram for this assignment.

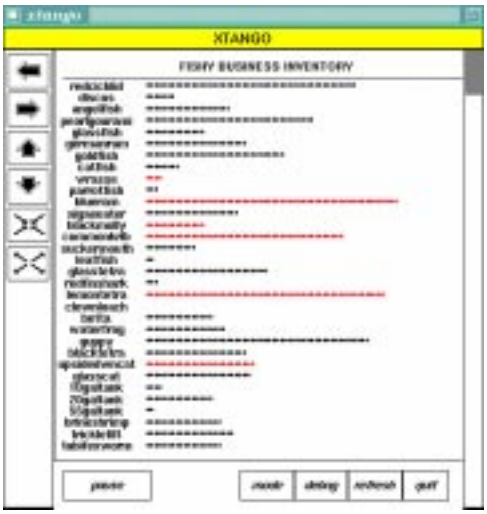


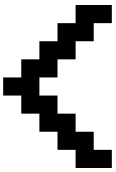
Figure 5: Snapshot of Histogram

Processing the tick marks was the most difficult part of this assignment since there were many cases to consider depending on whether the tick marks were red or black initially. With Xtango, processing of the tick marks is animated, showing tick marks appear and disappear as items are bought and sold. The animation can be slowed down when debugging to see each tick mark placed, and sped up once the assignment is complete. Many students struggled at first trying to debug the large datafile given to them, finally seeing the need for testing a small data file of cases.

6 Manipulating Bitmap Images

The final project we have used for the past four years in our CS 1 course is a program that manipulates bitmap images. Features of the assignment outlined below include introduction of two-dimensional arrays, reinforcement of control constructs and one-dimensional arrays, data compression, restoration of blurred images, and image expansion. To display images we incorporate the shareware program XV which is called by code provided to students. Images are stored in *portable bit map* (pbm) format, or as gray-scale (pgm) images. For example, the matrix of zeros and ones below on the left represents the image shown on the right although the image is enlarged eight times.

```
0 0 0 0 0 1 1 0
0 0 0 0 1 1 0 0
0 0 0 1 1 0 0 0
0 0 1 1 0 0 0 0
0 1 1 0 0 0 0 0
0 0 1 1 0 0 0 0
0 0 0 1 1 0 0 0
0 0 0 0 1 1 0 0
0 0 0 0 0 1 1 0
```



6.1 The Skeleton Program

Students are provided with a C++ class declaration for a class *PixMap*.[2] Data members of an object include the dimensions of an image and a two-dimensional array to store the image. Member functions are used to modify the image, e.g., by expansion, reflection, inversion, or enhancement. Although we supply the initial class design, good implementations require the declaration and use of other member functions whose specifications we do not provide, so the best students do participate in the design of this class to a degree.

Unlike the projects discussed above, images in this program are static rather than animated. However, the user interacts with the images using a menu of options that determine how the image is changed, e.g., by inversion, enhancement, rotation, and so on. A similar approach to classroom use of images is reported in [3]. Multiple images can be loaded, manipulated, and then saved although only one image is displayed at once.[3]

Students design and implement this menu-driven program from scratch. However, in a previous assignment (not de-

[2] We also have a complete, Java based version of this assignment.

[3] In our Java version of this program multiple images are displayed and the user selects the image that is manipulated.

scribed in this paper) we use another menu-driven program for which we supply the design (students implement the design). Students are expected to adapt that program in the design and implementation of this project. We want them to recognize the commonality in the design patterns of these programs. This is an example of conceptual re-use.

6.2 Data Compression

Since we use ASCII zeros and ones to store images in the *pbm* format, almost every black-and-white image can be compressed significantly using a simple scheme of run-length encoding. We explain how a sequence of zeros (or ones) is replaced by the count of how many times the zero (or one) occurs. Students understand this simple scheme and are able to implement both reading and writing of run-length compressed files. We provide a mystery image that is compressed using run-length encoding. When students succeed in implementing the function that reads compressed images, they can then display the image. Unravelling this mystery image is a surprisingly effective technique for getting students started on the project early.

6.3 Enhancing an image

Sometimes an image can be noisy, e.g., it might be garbled during transmission from an outside source. As an optional part of the assignment meant to challenge better students we ask them to implement image enhancement using median-filtering. Using median-filtering, the value of a pixel is replaced by the median value of its neighbors. Computing the median requires finding the values in an $n \times n$ neighborhood and then sorting. Of course edge cases require special care since there will be fewer than n^2 neighbors. This requires using an extra matrix to store the median values which are then copied back to the original image when the median-filtering has finished. This is necessary so that the pixels are replaced by median values from the original image, not from the partially reconstructed and filtered image.

Applying a 5×5 median-filter to the image on the left below results in the image on the right.

7 Conclusion

We have found that animations and interactive graphics generate student interest and enthusiasm which usually translates into better comprehension and mastery of the material in our courses. In addition, the visual component of the animations offers another dimension that assists in debugging, a task with which students have great difficulty and one we struggle to teach. Although the use of Xtango and Samba restricts these assignments to environments supporting X-windows, Lambada makes our animations accessible on all platforms running Java. The features of XV we use can be simulated using the platform independent graphics library reported in [6] which is C-based, but has recently been ported to a C++ class [10].

Assignments are available at our website.

`http://www.cs.duke.edu/csed`

References

[1] ASTRACHAN, O. *A Computer Science Tapestry: Exploring Computer Science and Programming with C++*. McGraw-Hill, 1997.

[2] ASTRACHAN, O., SELBY, T., AND UNGER, J. An object-oriented, apprenticeship approach to data structures using simulation. In *Proceedings of the Twenty-Sixth Frontiers in Education* (1996), pp. 130–134.

[3] FELL, H. J., AND PROULX, V. K. Exploring martian planetary images: C++ exercises for CS1. In *The Papers of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education* (February 1997), ACM Press, pp. 30–34.

[4] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] PATTIS, R. E. A Philosophy and Example of CS-1 Programming Projects. In *The Papers of the Twenty-first SIGCSE Technical Symposium on Computer Science Education* (1991), ACM Press, pp. 34–39. SIGCSE Bulletin V. 23 N. 1.

[6] ROBERTS, E. S. A C-based graphics library for CS1. In *The Papers of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education* (March 1995), ACM Press, pp. 163–167. SIGCSE Bulletin V. 27 N 1.

[7] RODGER, S. H. Integrating animations into courses. In *ACM SIGCSE/SIGCUE Conference on Integrating Technology in Computer Science Education (Barcelona)* (1996), pp. 72–74.

[8] STASKO, J. Tango: A framework and system for algorithm animation. *IEEE Computer* (1990), 27–39.

[9] STASKO, J. Using student-built algorithm animations as learning aids. In *The Papers of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education* (1997), pp. 25–29.

[10] TURNER, A. J. C++ port of Eric Robert's graphic package. http://www.cs.clemson.edu/∼turner, 1996.