Animating Algorithms: Project Progress Report

Steven Cooper (sjc209), Graham Le Page (gpl26),
Zhan Li (zrll2), Robert McQueen (ram48),
Andrew Medworth (am502), Neofytos Mylona (nm314),
Sidath Senanayake (sas58), Alan Treanor (ajit2)

12th February 2004

Contents

1 Introduction

This is the progress report document for the "Animating Algorithms" (Topic 8.5) Computer Science Tripos Part IB / Part II (General) group project.The authors are the members of group "Alpha", listed individually above. This project is being overseen by Steven Hand (smh22@cam.ac.uk).

2 Changes to specification

Algorithm Package

- API changed to move provision of initial data and animator moved to constructors instead.

Vector Animator

- `Markers` and `Pointers` merged into `Arrows` for ease of implementation.
- Copy element from one pointer to another method changed to take offsets rather than pointers.
- Removed vector primitive to compare two elements; unnecessary since you can just flash arrows.
- Delete element method removed; variable vector lengths aren't necessary for the sorting algorithms we plan to implement.
- Split vector method also removed for similar reasons; also hard to implement.
- Boolean highlight of `Arrows` supplemented with a few seconds of flashing (for both arrows and vector elements).

Shell Package

- It's not possible to use introspection to enumerate classes within a package, due to the way the class loaders work. Instead, an `AlgorithmCatalog` interface will be created (see below).

3 Code status

The latest copy of our code is available from our CVS repository, which is viewable on-line at:

`http://ned.ucam.org/cgi-bin/viewcvs.cgi/alpha/src/org/ucam/ned/teamalpha/`

This section contains a brief report from the authors of each package.

3.1 Algorithms (Sid)

All five of the proposed algorithms have been implemented and tested. More testing (especially of the graph algorithms) will be required by other testers/coders to ensure that the implementations aren't flawed. There are also prepared some sample connectivity matrices that can be used by testers; these are stored in the folder TestData under the file ConnectivityMatrices.txt. The code produced the correct results with this data.

Most of the algorithms have been commented at points where animator primitives will be called. During the integration phase, these comments will be replaced with the appropriate calls to animation methods from the `GraphAnimator` or `VectorAnimator` classes.

3.2 Animators

For both the graph and vector animator APIs, a set of exceptions for invalid input to the methods must be created, along with static functions which perform these tests. These tests can be used by all implementers of the APIs, to ensure a consistent response to invalid data - this consistent implementation is especially important to ensure that the queue does not accept an animation event which the actual animator will later reject.

3.2.1 Graph animator (Steven)

Almost complete, only highlighting nodes and edges and algorithm step messages have to be completed. The algorithm step messages require calling methods on the shell. The node and edge data is stored in a list of objects and a matrix of objects respectively. These have methods similar to the API calls which are called by the actual API method implementations. The class uses the same animation system as the Vector Animator, with a global timer picking events off an event queue, and causing the frame to be redrawn. Only used elementary tests to confirm various methods are working correctly.

3.2.2 Vector animator (Andrew)

The vast majority of this class has now been completed. It has been subjected to cursory, informal testing, including a preliminary linkup with Rob's queue class.

It was decided to break each animation primitive into even simpler animation events (e.g. moving an element out of its slot to lie beside the vector, moving an element vertically in this "channel", etc) which would then be queued internally by the animator and executed sequentially. This makes the system extensible, as there is a single global `Timer` which sends periodic `ActionEvents` to the

animator to tell it to draw the next frame, at which point a central method (`actionPerformed()`) looks at the current animation event (represented by an instance of the `AnimationEvent` inner class) to see what should be done next. It is a fairly complicated design, especially given that `waits` and `notifys` have been added to prevent animation primitive methods from exiting until the relevant animation has been completed (as is the intended behaviour for the API). However, it works and is still fairly easy to understand, and this, coupled with its extensibility, mean that we will not be changing this fundamental design.

The following features have yet to be implemented:

- Moving elements between separate vectors
- Highlighting of vector columns (for Radix sort)

There are also likely to be a few more small changes to the API (for instance adding back in support for a boolean method `highlight()` on vectors and arrows to indicate a temporary change of colour) and these facilities also have yet to be coded, although the implementation is likely to be fairly simple. There *may* also be an addition which allows the algorithm designer to specify whether an arrow is to be placed on the left or right of a vector: again, the implementation of this will not be difficult, because at present there is a boolean field in each `Arrow` object which determines this property (at the moment, it is always set to `false`, so that all all arrows appear on the right of their vectors): all that is required is to allow the user to set this at the time of arrow creation (it will not be possible to alter this property after creation time).

A problem has also been discovered with saving and restoring the animator state, when this is done by the queue in order to roll back to a previous algorithm step (checkpoint). The problem occurs when a new vector has been created since the last checkpoint. If the queue then attempts to restore the animator to the state before the vector was created, and run the algorithm again from that point, the `createVector()` method will be called again to create the same vector. This would not be a problem, were it not for the fact that the animator decides where to place new vectors on the basis of an integer field, `highestColUsed`, which tracks how many vectors there currently are on the canvas. The animator places the new vector to the right of what it thinks is the right-most vector: unfortunately, if the state is rolled back, this variable is not restored to its value at the time of the checkpoint, so the new vector is drawn to the right of its previous position and the whole restore mechanism breaks.

This can be remedied by using a more advanced way of laying out vectors on screen, which will keep track of which columns are vacant and insert newly-created vectors into the left-most free column. This will also solve a problem related to the deletion of vectors and arrows: at present, the space occupied by a deleted vector will be permanently left blank, and cannot be filled by another vector.

Finally, there is a more fundamental problem with this class in that it is not easy to change the format or layout of the canvas. For example, it would be difficult to change the width of vectors, or the size of the text, or the height of elements, or the gap between vectors, without altering myriad numbers within the class. If time permits, the code should be tidied up to put basic dimensions in global variables, to make them easier to change.

More testing needs to be done on this class: all animation primitives need to be exhaustively tested, especially the ones involving moving arrows very close together (it is unclear whether they will be redrawn correctly). Input of extreme data (e.g. very long vectors, large vector elements, or long arrow labels) is also a concern. The person responsible for the testing of this class is Alan.

3.3 Queues (Rob)

A class `GenericQueue` has been written to provide the common functionality which is used by both the `GraphQueue` and `VectorQueue` classes. As well as implementing the `AnimatorQueue` interface itself, it also provides an `enqueue()` function which allows the graph and vector queue classes to queue a method call, with the subject object, method name, arguments, and if one is necessary, the placeholder object that the queue will return from the method call. The `GenericQueue` stores these values as members in an internal `Event` class, and stores them in a linked list within the `State` inner class. When the algorithm invokes the `saveState()` method on the queue, a new `State` object is created, and the subsequent events stored inside that. When `next()` is called, the queue saves the state of the real animator, and then for each state in the linked list, uses introspection to find and invoke the methods on the actual objects. A hash map is maintained of placeholder objects which the queue returned to the algorithm, mapping to actual objects which the animator returned to the queue. Calls to the `prev()` method load the saved states back into the animator.

The queue classes are essentially complete, but further consideration will be given to making the queue class safe for concurrent operation, because it is intended to have the queue invoke the animator in a separate thread to the user interface code itself. As stated above, the queue will also need to perform the same checks on input data that the animators do, to ensure that no method call which is accepted and enqueued by the queue is later rejected by the animator. The queue will need to be tested more extensively with use of test algorithms and animators, as well as with the actual algorithms and animators (initial testing has already shown one odd behaviour with the positioning of re-created vectors in the vector animator).

3.4 Shell

The user interface of the shell has been designed on paper, and proof of concept code has been created as a basis for the actual implementation. There will be

three main screens in the system:

1. *The algorithm selection screen* (Diagram 1): this will be the screen that appears first when the system is loaded. It will have a list of available algorithms, with a description of each algorithm showing when the algorithm is selected. There will be a 'Next' button which when clicked will advance to the data input screen.

2. *The data input screen* (Diagram 2): this will have some text boxes for user input, the number of boxes will depend on the algorithm used. It will have a 'Random' button to generate random input, a 'Back' button to return to the algorithm select menu, and two buttons to start the algorithm. The 'Run Automatically' button will start the algorithm and run it through to completion, the 'Run Manually' button will animate the algorithm one step at a time, with the user choosing when to move on.

3. *The animation screen* (Diagram 3): this will show the animation of the algorithm, along with text areas containing description of what is happening at each step. There will be 'Next', 'Previous', 'Play' and 'Pause' buttons to step through the algorithm, as well as a 'Restart' button to return to the first step, and an 'Exit' button to return to the algorithm select screen.

In place of the introspection, which hasn't proved possible, the shell will internally make use of a new `AlgorithmCatalog` interface, which provides methods to enumerate the available algorithms, load them into the JVM, and instantiate the appropriate data-gathering, `Queue` and `Animator` classes for the shell, performing an abstract factory role to hide the details of graph vs vector algorithms from the shell. This will initially be implemented as a static catalog, which will contain lists of the initially available algorithms and their necessary queues and animators, but the same interface could be implemented by various class loaders, such as investigating a directory on disk, a configuration file, or even downloading algorithms from a web server.

4 Testing (Alan and Richard)

The testing of the project is governed by a single bottom-up strategy. All combinations of modules will be tested, starting with the individual modules and working up to the entire program. The strategy can be divided into two stages though: modular and integration testing, the former being the testing of modules individually and the latter the testing of modules that have been linked. Work has started on test animators and algorithms to exercise and implement all of the animation primitives in both the graph and vector animator APIs. A variety of test programs can be made which will run the test algorithms

against the real animators, with and without queues, to test that the animators function and correctly interact with the queue, and the test algorithms against the test animators via the queue, to independently verify the queue's correct functionality. Only recently has enough of the coding been completed to allow modules to be tested at an external interface level, so much of the testing has not been started. The modular testing thus far has taken a largely informal approach, with coders testing snippets of code as they develop a module. Test harnesses have been developed for each of the modules (some are yet to be completed) and these are to be used to test each of the modules in isolation. When all modules have passed their individual tests they will be tested together. Testing will not advance to the next stage, e.g. from testing pairs of modules to testing them in threes, until every test is passed. At each stage, bugs that are identified by the testers will be referred back to coders in a detailed code report, unless they are of such a simple nature, that it is more efficient for the tester himself to fix them himself.

For each erroneous input data case which has been conceived, a test case will be created to ensure that the error is caught consistently at the interfaces between modules. Special care will be taken to test that the queue and animators respond to erroneous input in the same way, to prevent problems with the queue failing to deliver events to the animator which appeared valid when called by the algorithm. The actual algorithms which have been implemented so far do not yet call animation primitives, so can be independently tested with the same inputs, to ensure correct action in all cases.

5 Documentation (Graham and Neofytos)

The basic user guide has been written, but will be updated in more detail as the project takes shape. The developers guide for adding new algorithms to the system is also being written, again this will be updated in more detail as the project progresses. Current copies of both have been submitted. All code, especially APIs which are implemented by multiple classes, is being documented by the coders as it has been written.

6 Plan of action

To be completed by Sunday 15th Feb:

- Finish off last parts of
  - algorithms - Sid
  - animator - Steven and Andrew
  - queue and threading - Rob

To be completed by Wednesday 18th Feb:

- Write and test shell - Rob and Sid
- Fully test modules - Richard and Alan

To be completed by Wednesday 25th Feb:

- Integrate all modules together - Sid, Steven, Andrew and Rob
- Fully test entire program - Richard and Alan
- Complete User Guide - Graham
- Complete Developers Guide - Neofytos
- Write final report - Graham
- Write individual reports - Everyone

Important dates:

- Thursday 26th Feb - Hand in final report
- Friday 27th Feb - Final review meeting with supervisor
- Monday 1st Mar - Deadline for code completion
- Wednesday 3rd Mar - Group presentation