University of Glasgow | School of Computing Science

Algorithm Animator

Arthur Bigeard
Alexander Ferguson
Andrew Gibson
Gediminas Leikus
Liam Bell

Level 3 Project — March 11, 2013

Abstract

For teaching purposes it is useful to be able to animate algorithms and produce a visual representation of how they work. The basic idea is to use a diagrammatic representation of a data structure, for example an array or a tree, and illustrate the algorithm step by step, showing how the data structure is accessed and changed. The aim of this project is to design and implement a system for animating algorithms. There are at least two possible approaches. One is to design and implement a simple programming language in such a way that all programs are animated while being executed. Another is to design and implement an API for animations, so that an existing program (in Java, for example) can be animated by inserting calls to your library. The system should be as general as possible in the sense of supporting a range of styles of algorithm, and should be demonstrated by producing a range of animations of standard algorithms. It would also be useful to be able to capture the animation in a form that can be viewed independently of your system, for example as a sequence of HTML pages or a Flash animation.

Education Use Consent

We hereby give our permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

Alice was beginning to get very tired of sitting by her sister on the bank and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it.

Alice opened the door and found that it led into a small passage, not much larger than a rat-hole: she knelt down and looked along the passage into the loveliest garden you ever saw.

Figure 1.1: Behind it was a little door

Chapter 2

Design

The following diagrams (especially figure) illustrate the process...

Chapter 3

Implementation

In this chapter, we describe how the implemented the system.

3.1 User Interface

Blah blah blah Blah blah blah Blah blah blah Blah blah blah

3.2 Problems Encountered

3.2.1 Arthur Bigeard

3.2.2 Alexander Ferguson

3.2.3 Andrew Gibson

3.2.4 Gediminas Leikus

Separating animations into steps and making continuous animations

The group decided to use Java for our project, because we were all familiar with it and because it was the language our client suggested us to use. We had left the decision of choosing the animation tools to Andrew Gibson and Alexander Ferguson. For the reasons stated in this document (reference), they decided to choose the Timing Framework [1].

Andrew had an example (3.1) ready, which had multiple rectangles created from an array moving back and forth, had buttons allowing us to stop the animation, resume it or delete one of the rectangles. He also added comments into the sample code, just so it would be easier for us to familiarize with the tool.

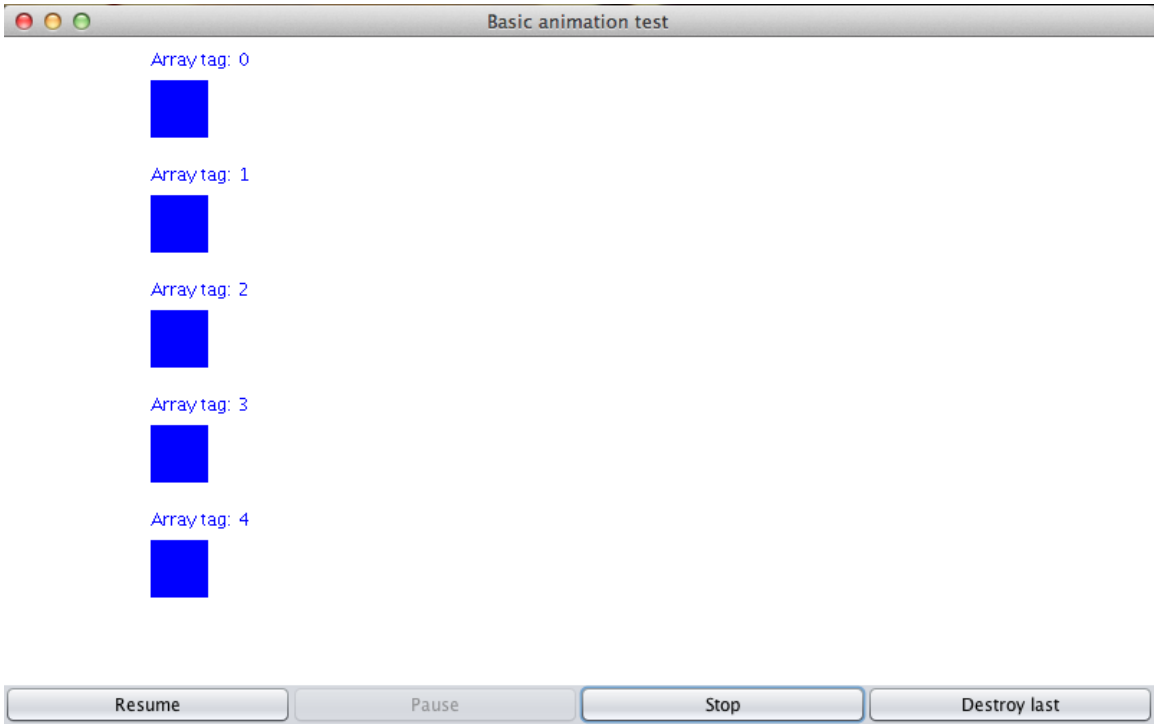

Figure 3.1: The first demonstration, produced by Andrew

While this was a good initial step, we did not have any obvious way to implement Steps, in a sense where multiple Animations could be assigned to one step and we could navigate through these steps back and forth. Therefore both Andrew and Arthur Bigeard started looking into it.

Andrew had no real suggestions and Arthurs prototype implementation seemed:

- too complex (a swap of 2 rectangles was over 50 lines long 3.2)
- impractical to use (it didnt have any way to step back and forth through the animation)
- it had a performance overhead (all Animator objects were running continuously, until the whole animation was stopped)

It had a few good points though:

- it was done on a low level (so the how-to part was obvious if you understood the code)
- it was not storing too much information (thus it did not really have a memory overhead).

But this was not what we wanted the most important thing for us was the stepping back and forth functionality. Therefore I started looking into it as well.

During my research of the Timing Framework I found out that we could add triggers, which would start an animation, to a button or to another animation, which would start another animation when

```
public void timingEvent(Animator source, double fraction){
    boolean done = false;
    if(anArr.getToDo().size() > 0){
        Step[] steps = new Step[anArr.getToDo().peek().length];
        System.arraycopy(anArr.getToDo().peek(), 0, steps, 0, steps.length);

        for(int i = 0; i < steps.length; i++){
            Rectangle rect = rect_list[steps[i].getIndex()].getRec();
            double x = steps[i].getX();
            double y = steps[i].getY();
            double stepX = rect.getX();
            double stepY = rect.getY();
            if(y != stepY){
                if(Math.abs(x - stepX) < 50){
                    if(y > stepY){
                        rect_list[steps[i].getIndex()].getRec().x += 1;
                    }
                    else{
                        rect_list[steps[i].getIndex()].getRec().x -= 1;
                    }
                }
                else{
                    if(y > stepY){
                        rect_list[steps[i].getIndex()].getRec().y += 1;
                    }
                    else{
                        rect_list[steps[i].getIndex()].getRec().y -= 1;
                    }
                }
            }
            else{
                if(x > stepX){
                    rect_list[steps[i].getIndex()].getRec().x += 1;
                }
                else{
                    rect_list[steps[i].getIndex()].getRec().x -= 1;
                }
            }
            if(y == stepY && x == stepX){
                done = true;
                anArr.getToDo().poll();
                anArr.getDone().add(steps);
                break;
            }
        }
        repaint();

    }
```

Figure 3.2: Arthur example code making an animated change in coordinates

```
s.addAnimator(new Animator.Builder().setDuration(time, TimeUnit.MILLISECONDS)
.build(), nextBtn);

s.getLastAnimator().addTarget(PropertySetter.getTarget(rect_list.get(b),
"currentX", rect_list.get(b).getRec().x, rect_list.get(a).getRec().x));}
```

Figure 3.3: My example code making an animated change in coordinates

it stops or starts running [2]. Thus, triggers seemed like a great tool for linking multiple animations into one Step, linking steps and making a continuous animation.

I still wanted to find a way to make a change of a rectangle (or any other object) property easy to manage. Looking through the Demos I also found that I could use Targets and PropertySetters to do this in just a few lines of code 3.3, which seemed great. But it only allowed us to change a property, which was numerical (like coordinates or colors) it didnt allow us to change all the properties of the object, like the labels of rectangles. To solve this issue, I researched further and found that we can create a class, which would implement a TimingTarget interface and its begin(), end(), repeat() and reverse() methods. Therefore, to solve the issue of changing the labels of rectangles I created a new class called ChangeLabel, which implemented a TimingTarget interface, its begin(), end(), repeat() and reverse() methods, and added a change of String variable just inside its begin method. This approach seemed great, because:

- we can change any property of an object or do pretty much anything when that TimingTarget is called
- it is attached to each Animator object, so we do not have to worry about storing the Timing-Targets anywhere
- it is efficient performance wise, because these TimingTargets are only executed, when the Animator object is
- it is easy to use, implement and understand

Therefore, the end result of my prototype was:

- a list of steps
- an array of rectangles and their coordinates at each step
- an ability to either step back and forth through the animation by using buttons or see a continuous animation (and it was either that or that)

After looking through both my and Arthurs prototypes, we decided to merge the good parts of each: we kept most of my prototype, but reduced the amount of data stored for each Step, in particular, we made it so it would only store the details of the changed object before the change, rather than the details of all the objects.

The only issue left then, was the ability to have both step-by-step and continuous animations and allow the user to switch between them at any point of time, but this was not a big issue, since I just used a similar approach I used with ChangeLabel class:

```
27      @Override
28      public void begin(Animator source) {
29          // TODO Auto-generated method stub
30          if (reference instanceof Rect) {
31                  ((Rect)reference).setLabel(labelTo);
32          } else if (reference instanceof AnimatedArray) {
33                  AnimatedArray.setInfo(labelTo);
34          }
35          AnimatedArray.panel.repaint();
36      }
37
```

Figure 3.4: ChangeLabel begin method dependency on AnimatedArray

1. I created a new class, called ContinuousAnimation, which was also implementing the TimingTarget interface
2. introduced a boolean variable called continuousAnimation, which was keeping record of whether the animation was in step-by-step or continuous mode
3. made it so that the begin method in ContinuousAnimation would execute different actions according to the boolean continuousAnimation variable value:
    - If true, Trigger the next Step
    - If false, do nothing

Refactoring

Our first common implementation that we were using for development was coupled too much with AnimatedArray data structure and was not flexible at all. This became obvious, when we started implementing our AnimatedLinkedList. When we wanted to make our API to animate an AnimatedLinkedList instead of AnimatedArray, we had to change multiple classes, including Rect, ContinuousAnimation and ChangeLabel 3.4. This was due to no polymorphism in our design.

After doing our Professional Software Development 3 (PSD3) D7 deliverable (which was implementation of an Internship Management System we were designing the entire first semester), I have learned how to structure software better and how we can split code into components in Java. Therefore it was time for refactoring.

One of the first things I did was to create a separate package for each data structure. This lead to 4 additional packages being created:

- AnimatedArray
- AnimatedBinaryTree
- AnimatedDataStructure

```
11  11  public class Main {
12  12
13      -   private static AnimatedLinkedList anim;
        13  +   private static AnimatedArray anim;
14  14      public static int time = 200;
15  15
16  16    public static void main(String[] args) {
17  17      int[] arr = {4,3,8,1,2,12};
18      -   anim = new AnimatedLinkedList(arr);
        18  +   anim = new AnimatedArray(arr);
```

Figure 3.5: Changing animation to animate a different data structure

- AnimatedLinkedList

Each package is supposed to keep classes relevant only to its data structure, apart from Animated-DataStructure package, which keeps classes that are relevant to all data structures.

I also completely separated the GUI from the rest of the code and created a class called setupGUI.

The next thing I did was to create a public interface AnimatedDataStructure, which is how we solved the issue I mentioned in the first paragraph. Rect, ContinuousAnimation, ChangeLabel and setupGUI are all expecting an AnimatedDataStructure variable now and all data structures are just implementing this interface. Therefore it does not matter which data structure we use, the common classes (which are in AnimatedDataStructure package) work for all of them. The interface itself also brought common method names and some structure. Another good thing is that now, we only need to change 2 things in the Main class if we want to work with a different data structure: the data structure class name and then the constructor method call 3.5. This did bring in some repetitive methods, like getNextButton(), which are being repeated in all data structures, but could be solved by introducing another abstract class, which would be implementing the AnimatedDataStructure interface and have all of these repetitive methods and then making all data structure classes extend this new abstract class.

All of the above solutions introduced basic polymorphism to our software and solved quite a lot of development issues as well as made it easier to develop and manage the code for future developers (if there are going to be any). While doing all of the above, bits of unneccessary code were removed or changed as well.

3.2.5 Liam Bell

Chapter 4

Evaluation

We evaluated the project by...

Chapter 5

Conclusion

ASD

5.1 Contributions

5.1.1 Arthur Bigeard

5.1.2 Alexander Ferguson

5.1.3 Andrew Gibson

5.1.4 Gediminas Leikus

5.1.5 Liam Bell

Bibliography

[1] Timing Framework. Official website. http://java.net/projects/timingframework, 2013. [Online; accessed 4-March-2013].

[2] Timing Framework. Official website, demos page. http://java.net/projects/timingframework/pages/Demos#Triggers, 2013. [Online; accessed 4-March-2013].