1 Algorithmic Animation in Education

1.1 The motivation for algorithmic animation

As mentioned in the introduction, algorithms are well-defined sequences of computations which transform input data into output data. A formal definition of *algorithm* always makes reference to a specific computational model, such as the lambda calculus or the Turing machine. The central aspect of any algorithm is that it involves *computations* and thus determines a specific trajectory in the state space of machine configurations [Sedgewick 03, Cormen 90]. In imperative computer languages, machine state transformations are handled explicitly in the code and affect specific data structures such as single variables, arrays, lists, or trees. Understanding an algorithm often involves reconstructing the machine state evolution with the mind's eye. Debuggers, which allow us to follow the sequential execution of program code and to define breakpoints, are useful for understanding a program and for experimenting with it.[2] But many educators think that animating an algorithm can provide students with a deeper level of comprehension, which goes beyond just watching debugging steps. For this reason, some authors have extended debuggers with algorithm visualization capabilities [Mukherjea 94]. My own interest in algorithmic animation started long ago with the work I did for simulating and visualizing the operation of a parallel Prolog machine [Esponda 87].

Even good formatting of a program can make a difference to its comprehendability. An active community of researchers has studied the relationship between readability of software and different formatting strategies for programs. Good annotated and formatted programs are easier to understand and maintain [Knuth 84]. The eye can sometimes help to organize the code – we profit from its natural ability to categorize objects in groups [Oman 90]. We will return to this issue, from the point of view of visual design, in Chapter 3.

---

[2] Declarative languages have seldom been visualized. Functional languages, such as Miranda or Haskell, do not even provide a debugger. The whole idea of a declarative computation is to eliminate machine state space thinking: the programmer specifies the relationship between input and output, for example in logical terms, and the computer takes care of providing the necessary steps. In reality, much state space thinking is used to write programs in functional or logic programming languages, such as for example Prolog.

Eventually, what an educator wants is not just to make an algorithm easy to understand during a class, but also *easy to remember*. The student should be able to recall the mechanics of the algorithm after having learnt it, maybe just once. As mentioned before, debuggers are effective for stepping through code thus making it more readable, but they are not effective in *imprinting* the algorithm's steps on the students' minds. For this, an algorithm must be considered from a more abstract perspective, from the point of view of *abstract data types*. According to [Lohse 94], visual representations are also "data structures for expressing knowledge." Many psychologists are of the opinion that thinking involves and requires images [Arnheim 69], and we now talk of "visual literacy" as a skill that can be trained [Dondis 73].

Images for computer science concepts require models. Abstract data types are defined through the operations possible on the data, regardless of the concrete implementation. A stack, for example, is defined through the *push*, *pop* and *empty* operations. Visualizing an algorithm which makes use of a stack requires providing a physical visual model for the operations and the substrate they act upon, for example, a pile of books, one on top of the other. A *model* is the flesh and bones of any algorithmic animation. A model is something which can be understood intuitively by the viewer, something related to our visual or physical experience.

Therefore, the main problem for a successful algorithm animation is to select the appropriate visual model for the abstract data types involved in the computation and the appropriate level of aggregation of the algorithm's steps. The execution granularity determines if we will be able to see only high-level operations or also the atomic manipulations that occur in each step. Ideally, an algorithmic animation system should provide both possibilities through user control of the animation granularity.

The right visual model is the single most important aspect for the production of a high-quality animation. A visual model should make the algorithm's operation understandable just by producing pictures of the abstract data types and the transitions involved. Mathematical proofs are a good example of the kind of techniques required in algorithmic animation.

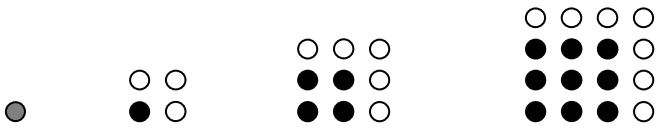

Figure 1.1 Proof without words: the sum of successive odd integers is the square of an integer

In Mathematics there is a long tradition of producing images for "proofs without words". The statement of a theorem and a picture are provided and the reader can

"see" immediately that the theorem is true. In *proofs without words* the model fits like a glove to the semantics of the theorem. Consider for example the sequence consisting of the sum of successive odd numbers: 1, 1+3, 1+3+5, 1+3+5+7, and so on. It is easy to prove that every member in this sequence (every partial sum) is the square of an integer. Just look at Figure 1.1, a proof without words.

Interestingly, in Mathematics, proofs without words are *static pictures* of objects adorned with a few labels. A little reflection on the part of the reader suffices to convince her that the proof in Figure 1.1 is correct [Nelsen 1997].

In computer science, on the other hand, we are interested in dynamic data structures being modified by an algorithm. A single picture is usually not enough: *sequences of pictures* or a *smooth animation* are needed to make the algorithmic transformations come alive. However, the purpose of the animation is the same as in mathematical proofs without words: the essence of the method being discussed should be coupled to the human visual channel, so as to make the method evident and also *memorable* for the student. The best algorithmic animations are those that cannot be forgotten, because they are so simple and natural, that they fit the algorithm perfectly.

Let me give another example from Mathematics which can help to illustrate this idea. There is a classical proof of the Pythagoras Theorem that relies in just two squares drawn inside one another. A triangle with sides a and b and hypotenuse c repeats four times inside the larger square (see Figure 1.2).

From the diagram it is evident that $(a+b)^2 = 2ab + c^2$. Pythagoras Theorem follows immediately from this identity.

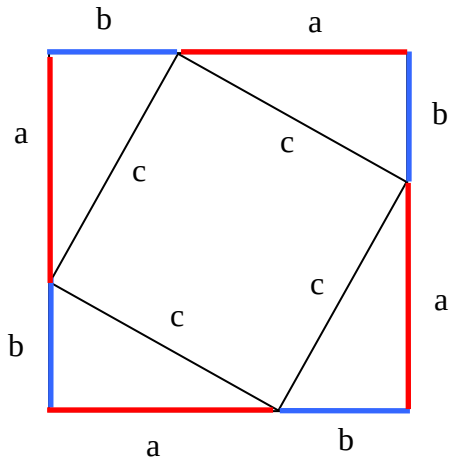

Figure 1.2 Proof without words of Pythagoras' Theorem

It could be that in the visual proof the validity of the theorem is just an accident, just valid for the particular *a*-*b*-*c* triangle used in the picture. A way of testing this,

would be to let *a* and *b* vary continuously over a certain interval and watch how the resulting picture transforms. This would amount to a proof by visual animation, which has been implemented by some authors, for example, using the Cinderella geometry system [Kortenkamp 99].

We have the same problem in algorithmic animation as in animated proof systems: it would be desirable to produce animations on demand, using data provided by the viewer. In this way, the correctness of the algorithm can be empirically tested with new data. Also, the variations in program behavior can further reinforce the understanding of the algorithm. Such computer science visual proofs using data structure visualizations have been advocated by Goodrich and Tamassia [Goodrich 98]. In their teaching practice they have found that students can recall the properties of data structures best, when they have learned to associate a picture with it. One of their examples is the proof that a heap can be built in linear time from an unordered array. They used a single picture for their visual proof.

1.2 Algorithmic animation as a subfield of software visualization

It is important to be aware of the relationship between algorithmic animation and other forms of software visualization. The diagram below, improved from the one shown in [Price 98], shows the different areas of specialization of "software visualizers". The first main subdivision has to do with the subject to be visualized: programs or algorithms. In the first case, concrete code for a certain programming language is involved and in the second, we are more interested in a high-level view of computation, like the algorithmic manipulation of data structures.

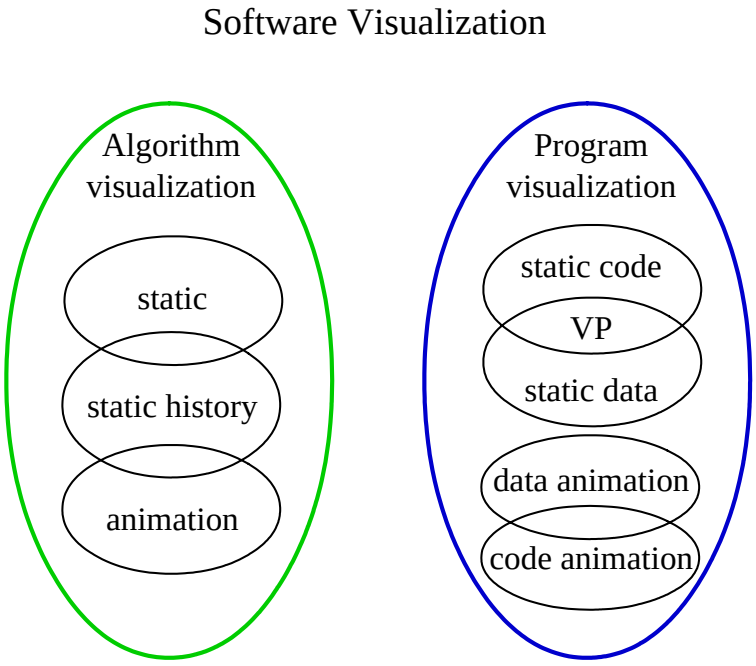

Figure 1.3 The subfields of Software Visualization

In the subfield of program visualization, both the code and the data can be visualized statically. Flow-charts, for example, provide an organized view of code [Scanlan 99]. Different variations of "literate programming" try to make code as readable as a well written book [Knuth 84]. Visual Programming (VP) is an attempt to offer the programmer a way of specifying computations using diagrams and direct interaction with a computer [Glinert 90, Chang 87]. Some program visualization tools offer the programmer the possibility of animating automatically code and data. Debuggers, for example, allow the programmer to step through the code, see it execute and transform monitored variables [Baecker 68]. In this thesis we will not deal with program visualization, we shall only concern ourselves with visualizing algorithms.

There are three possible levels of algorithm visualization. The first and third are mentioned by [Price 98], but not the second (static history). We can visualize an algorithm statically by seeing the pseudocode and a diagram of the data structure on which the algorithm operates. We can animate the pseudocode and a model of the data structure. This is the classical meaning of the term algorithmic animation ("algorithm animation is dynamic algorithm visualization," [Brown 98b]). But an intermediate possibility is to produce a history of pictures of the operation of the algorithm. This is the method used in books to illustrate algorithms, and is also one that we will explore later in this thesis. The new editions of Sedgewick's algorithms book, incorporate many such static histories of algorithm runs, to great effect. The dynamics of the sorting algorithms can be visualized to an extent that it is easy to recognize the kind of sorting algorithm being used alone from a scatter plot of the data being sorted [Sedgewick 03].

In this thesis, I look at the three possibilities shown on the left side of Figure 1.3 of visualizing algorithms and data structures: static, history, and full animation visualizations, using two different platforms: an electronic blackboard and the Macromedia Flash animation engine.

1.3 Academic experience with algorithmic animation

There are many visualizations of algorithmic animations available on-line. Most are used by students, as part of a course, and have been developed by university staff. Some universities have invested considerable resources in electronic classrooms where such animations are used [Bazik 98]. Animations have also been used in Artificial Intelligence for state space search [Ciesielski 01, Stern 97], genetic algorithms [Jackson 97], expert systems reasoning [Selig 90], visualizing knowledge based systems [Domingue 98], and even for understanding neural networks [Jackson 96]. Recently, algorithmic animations in Java have gained prominence in the Web. This shows that there is an eager audience for them. The effectiveness of program visualizations for educative purposes has been a recurrent topic [Badre 89, 92, Cunniff 87, Green 96, Rieber 89, 90a, 90b]. Neverthe-

less, not enough evaluations have been conducted about the pedagogical effect of animations on students' learning and some of the existing ones are contradictory.

One of the most careful evaluations was conducted at Auburn University [Hansen 1998]. Students were divided into an animation group and a control group. They learned the same algorithms, but in one group they were given an animation written with the Toolbook system (similar to Hypercard in the Macintosh) and in the other group, they used pages from a book. Both groups were academically equivalent, judging from pre-test examinations. The results of a post-test showed a positive effect of the animations on students' proficiency:

- The students in the animation group had 74% correct answers in the test, compared with 43% in the control group.
- When two sorting algorithms were learned, the percentage of correct answers were 63% and 44% for the animation and the text group, respectively.
- Only when the control group had to solve additional homework questions did the proficiency of both groups become comparable.

The main caveat for this study was the limited number of animations tested. Only four sorting algorithms and their differential learning in the two groups were compared.

Other researchers have not obtained so statistically significant and clear-cut results. On the one hand, it is difficult to teach complete courses dividing the students into test and control groups. On the other hand, there are many aspects that can contribute to obscure the comparison. It can be that students in the animation group, for example, become more engaged with the topic because of the novelty of the technology, whereas students in the control group perform at their normal level of attention. The quality of the explanation of the algorithms in each group can be also different. It is difficult to quantify how the engagement of the students with the technology evolves over time. More extensive studies are needed.

In contrast with the evaluation mentioned above, studies by Stasko and his colleagues [Stasko 93a] did not detect a significant advantage of animations in learning situations. The explanation they advance is that students may not make the mental connection between the animation itself and the abstract aspects of the algorithm. For an expert, these connections are obvious, while for a student they may not be. Remember the proof of the Pythagoras Theorem illustrated in the previous section. For a trained mathematician, the proof is obvious. For a new student some words would be needed to complement the "proof without words". It seems, therefore, that the level of expertise of the viewer should not be forgotten while designing an algorithmic animation.

[Byrne et al. 96] reached a conclusion along other lines. It seems that simple algorithms were learnt better by the group using animations, but complex algorithms are learnt equally well with or without animation. It could be that the visual chan-

nel is overwhelmed when the complexity of the animation exceeds a certain level of detail. Psychologists who have compared learning with pictures to learning with animations conclude that the value of an animation depends on the subject matter being studied and the amount of practice that the pupils obtain. An animation can engage the students more if the instructional environment is under their control [Rieber 90a].

Kehoe et al. examined the use of algorithmic animations by students from a different perspective: students were asked to solve homework problems and the animation was one of the materials that could be used. A control group had no access to the animations [Kehoe 01]. The animation group performed significantly better than the control group. A review of the way the students used the animations showed that most students worked with the animations as a way of figuring out better the mechanics of the algorithm. Based on this experiment the authors of the study advanced three hypotheses:

- The advantage of algorithm animations is more evident in homework situations, when the student can work with the animation to solve problems.
- Animations make an algorithm less intimidating, enhancing comprehension.
- Algorithm animation can facilitate learning of the procedural aspects of an algorithm.

Some learning researchers have postulated that learning is best when more than one communication channel is involved. An animation synchronized with an explanation is therefore better than just an animation followed, or preceded by an explanation. This is usually done by the lecturer in the classroom, when explaining the animation being run, but an oral explanation is not usually available in simulations provided on-line. Other researchers have tried to activate other channels, for example, using color to code animation objects, or using sound effects to make users “hear” their data or programs [DiGiano 92a, 92b, Francioni 91, Gaver 91]. My animation of a data structures learning unit (discussed in Chapter 3) combines the textual explanation with the animation. The animation itself makes textual explanations pop-up for better comprehension of the algorithms. As we will see later, my algorithmic animations for the electronic blackboard and in Flash can be enriched with sound.

The most comprehensive review of the educational experiments performed with computer animations was undertaken by Hundhausen, Douglas and Stasko [Hundhausen 02b]. They reviewed and compared 24 different educational experiments, especially the very extensive ones done by Lawrence [93, 94], trying to find out which methodology was applied and which kind of cognitive approach was being tested. The authors find that algorithmic animation has been used for seven main purposes:

- for use in lectures,
- for independent study,

- for homework,
- for discussions in class,
- for laboratories,
- for discussions in office hours,
- for tests.

The success reported had to do with the kind of application that was tested in each study. A global summary of the 24 studies was produced by the authors. The experiments reporting significant results (as a benefit of teaching with algorithmic visualization versus teaching without it) comprised 46% of the experiments, whereas in 42% of the experiments non-significant results were found.

Stasko [97] studied computer animations in the classroom following a completely different approach. He conducted pedagogical experiments in classes where the students had to write their own algorithm visualizations as part of the assignments. He concludes that students do indeed benefit from this activity. Building an animation is very much like having to teach the algorithm to another person. The student does not only implement the algorithm – he or she has to actively look for the best possible way of illustrating the algorithm steps. Not surprisingly, the best students in the class were also the ones who developed the best visualizations.

Hübscher-Younger [2003b] has confirmed with her own experiments that authoring animations can foster better understanding of algorithms. In her approach, students build their own visualizations, share them and evaluate the results. Pre and post-testing showed that students could profit from this exercise. The explanation advanced by the authors of the experiments is that when students build their own explanations, they are able, at the same time, to understand the concepts better. In these experiments the students were allowed to use any kind of visualization engine including paper. In previous experiments, HalVis, a Hypercard based system, and CAROUSEL, a Web based system, were used [Hübscher-Younger 03a]. Another author who has experimented with "active algorithm learning" is [Faltin 02]. His system present animations and ask questions, which the students answer by helping the animation to continue.

The success of these experiments probably validates the constructivist approach [Papert 80]. Knowledge is a construct of the mind. Students have to build their own conceptualizations and this is best accomplished when they have to externalize their own mental modeling using a computer visualization. Hall et al. found, for example, that students that draw their own diagrams of a physical process get higher scores on tests, than students who only have a text at their disposal or even a text with illustrations [Hall 97].

More studies are needed in order to fully understand the potential impact of computer built animations but one thing is clear: if students are to program their own animations, this should be easy to do. A cleverly designed system is needed,

which allows the students to obtain immediate feedback. *Automatic* algorithmic animation would be the answer but, for the reasons to be discussed in the next chapter, automatic animation remains still an unsolved problem.

Additionally to cognitive constructivism, two other theories of learning play a role in the experiments that have been performed to date: one is epistemic fidelity, the other the dual coding hypothesis. Epistemic fidelity assumes that humans have mental models of the physical and logical world, and if the visualization directly corresponds to or stimulates those models, then efficient learning transfer is achieved, i.e., the students learn the subject better. The dual coding hypothesis, on the other side, postulates that when information is coded in both a non-verbal and a verbal way, this dual code also makes the transfer from teacher to student more efficient [Hundhausen 02b]. As we will see in Chapter 3, probably all these different considerations play a significant role for the production of good algorithmic animations.

1.4 Some preliminary conclusions

Visualization is not the panacea for solving all problems students or programmers have understanding algorithms. Visual programming languages have not been as effective as initially thought because textual communication can be superior for certain complex tasks [Petre 93, 95]. Many programmers prefer to work, for example, with editors such as Emacs, instead of visually oriented ones. They prefer the versatility of the programming tools and the speed with which it is possible to define Emacs commands.

We must also distinguish between animations which have the purpose of allowing the viewer to absorb or handle large quantities of information, and those geared towards understanding the dynamics of algorithms. Experiments at Xerox with user interfaces, for example, have led to 3D visualization methods that can be used to see the desktop and directory hierarchies in novel ways [Robertson 93]. The intention of these systems is to shift the load from conscious cognition to subconscious perception.

The main conclusion that we can therefore extract regarding algorithmic visualizations is that they are *primarily for learning*. Although subtle bugs can be perceived and found by running an algorithmic visualization, that is not the purpose of the animation itself. While in visual debugging we are more concerned with catching all possible state transitions and being able to trace and retrace steps, in algorithmic animation we are trying to stimulate the visual channel of the viewer. We try to attract her attention to certain parts of the animation and make her disregard others. We are making her focus on certain steps which are the essential ones. Visual debugging is neutral in this respect: *everything* should be visible; a bug can be hidden in the important, but also in the not so important lines of code.

From his long experience with algorithmic animation, Stasko has drawn the conclusion that animations are one more of the instructional materials that teachers can use, but one that attracts the interest of students to algorithm analysis and design [Stasko 98b]. He forcefully makes the point, that any instructional material that gets the attention of the students is useful, because it keeps them engaged. Some kind of interaction is needed, though. An animation should not be conceived as a video: the students should have the possibility of producing their own animations or at least, of providing data for them.

Based on the material discussed in this chapter, I can now formulate the principles that guide the further research described in this thesis:

A) *Algorithmic animations should contribute to the creation of a mental model of the operation of an algorithm that will allow the viewer to recall the mechanics of the algorithm easily, both in the short and in the long term.*

B) *Algorithmic animations are for learning – they should be combined with textual or oral explanations of the mechanics of the algorithm.*

C) *Students should be able to produce their own animations with low overhead, in any desired programming language*.

D) *Animations should be distributable – students and lecturers should be able to export the animations to the Web or send them by e-mail, possibly in different formats*.

The next chapter provides a historical review of algorithmic animation, before proceeding to work towards the goals stated above. In Chapter 7, I rephrase the above ideas using Tufte's description of graphical excellence [Tufte 83].