ANIMAL-FARM: An Extensible Algorithm Visualization Framework

Guido Rößling
TU Darmstadt, Dept. of CS
D-64283 Darmstadt, Germany
+49-6151-165411
roessling@acm.org

ABSTRACT

Algorithm visualization systems have gained much interest lately. However, most of today's animation systems do not support reuse or the integration of extensions. If extensions can be implemented, they usually require source code modification and intimate system knowledge. In this paper, we present a simple yet powerful framework architecture for AV systems that allows easy reuse and extensibility, even at run-time.

Categories and Subject Descriptors

K.3.1 [**Computers and Education**]: Computer Uses in Education – *Computer-assisted instruction*

Keywords

Algorithm Visualization, Framework

1. INTRODUCTION

One central challenge in computer science lies in understanding the dynamics of algorithms and data structures. This is of particular importance in two areas: computer science education and programming, especially concerning debugging. Understanding both how and why an algorithm works is challenging for students and novice programmers. Detecting and fixing bugs can be difficult even for advanced programmers.

Software teaching novices how to program and debug their code is increasingly used in computer science education. Algorithm Visualization (AV) systems are a special application of software visualization, focusing on visualizing the dynamic behavior of software. The interest in AV from computer science education has steadily increased over the last years, as evidenced by the number of publications on the topic

A large number of AV systems are already available, mainly from educational institutions. They differ in a large number of ways, including the degree of interactivity, display strategy and the type of supported input. They also range from non-interactive slide show-like displays to full-fledged debuggers that let the user specify attribute values and invoke methods. A large variety of different display control elements exist between the two extremes. Finally, the input data used by AV systems ranges from actual source code in a specific programming language to interactive graphical editing using direct manipulation.

The large variety of available systems presents a formidable challenge for users interested in employing AV. The strengths, weaknesses and restrictions of a given system are usually not obvious at first glance. One drawback common to most current AV systems is the lack of support for easy extension and customization. It is highly unlikely that a given fixed system can meet all demands of future applications. However, most systems cannot easily be extended with additional features.

In this paper, we present an AV framework that is comparatively easy to implement. The framework is geared for flexible general-purpose AV systems that can in principle animate any type of algorithm. Several framework components can be added or removed on the fly. The input type is not fixed, allowing for a wide range of different animation generation approaches. Advanced controls including the reverse playing of steps are prepared in the framework and do not require much work by developers encoding AV systems based on the framework.

The paper is structured as follows. In section 2, we present some requirements for AV frameworks. Section 3 presents basic design issues of the framework, followed by the framework components in section 4. Section 5 concludes the paper and outlines areas of future research.

2. AV FRAMEWORK REQUIREMENTS

A framework consists of a set of cooperating classes that together represent a reusable design for a specific class of software [2]. In this chapter, we focus on motivating and describing a framework for algorithm visualization. The framework defines the basic architecture of all AV systems built using it. In general, it emphasizes the ability to reuse the basic design rather than individual classes or pieces of code.

Frameworks are built for reuse. Typically, a system developer will have to extend certain framework classes and follow implementation guidelines to build a concrete framework-based application. Frameworks usually contain fixed and ready-to-use components as well as abstract classes and interfaces that have to be filled with content by the developers. All AV systems based on the framework will therefore have the same core components and a similar structure of classes, making them easier to maintain and more consistent.

The downside of framework development is that specifying a good framework structure is very difficult. Significant amounts of work have to go into creating a framework structure that can be reused easily on the basis of its merits regarding conciseness, understandability and easy applicability. The benefit of the framework is that it can make developing new concrete implementations much easier.

One issue in specifying an AV framework is making it applicable for the large variety of different understandings of what constitutes AV. For example, there are large conceptual differences between topic-specific systems that illustrate a single algorithm or a limited topic area, and code interpretation-based systems that illustrate the actual behavior of the underlying code.

Therefore, the framework should not fix the animation generation approach.

Additionally, it is highly unlikely that any given AV system will be able to address all expectations of the target audience. The "standard" process in many areas of software development is to start developing a new system once a (foreign) system is found to be inadequate in some areas. Instead, it would be preferable if one could simply extend the other system to address the required functionality. The key word here is *simply*: usually, even if the source code of the system is available, adding extensions is only possible after a detailed study of the implementation. Additionally, the developer will usually have to modify parts of the (foreign) source code. In the sense of object-orientation, we therefore postulate that the framework should be easily extensible, based on a documentation of its structure – without requiring any deep system knowledge, and if possible also without modifying existing code.

There are a variety of additional requirements that the framework can already address, thus providing the functionality to all implementing concrete systems. First of all, a clean specification of the participating parties in animation helps developers implement the concrete systems, as well as extensions. If the main animation component classes are fixed, the framework can already provide a complete default implementation of the animation display front-end. This front-end has to cover advanced control functions, such as setting the magnification and speed of the animation display. A dynamic and static view of the individual animation steps shall also be incorporated.

3. FRAMEWORK DESIGN ISSUES

Two central design goals of the ANIMAL-FARM AV framework are extensibility and configurability. Thus, it shall be comparatively easy to implement new functionality for a framework-based system, ideally without having to touch the actual source code. Achieving this ambitious goal is not easy, regardless of whether we examine the issue from the AV angle or from software engineering.

We have isolated four target areas that need addressing in a dynamically extensible framework:

- Internationalization issues,
- Representation of object state,
- Dynamic component loading and configuration,
- Component decoupling.

Note that these requirements are not specific for AV usage, but apply to all conceivable extensible frameworks. Internationalization touches on the required support for translating the given system's GUI front-end on the fly to the selected target language. It is a powerful feature that can instantly make any system more attractive to prospective users, provided that their mother tongue is supported with good translations. However, while translating the user interface may be helpful, it is rendered ineffective if the actual content remains in a different language. Therefore, the framework shall also be able to address the translation of content. For this end, we have implemented a small set of classes in a separate package that allows the easy creation of translatable Swing-based GUI elements. In fact, the generation of Swing elements is easier using this package than coding them "normally", without internationalization support!

Both the translation of the front-end and the AV content relies on the presence of external files containing the translated content. Depending on which language resource files are present, the translation into an arbitrary number of languages is possible on the fly. Note that content translation is only possible if the AV components are laid out dynamically using relative coordinates, as the width of translated texts varies widely between languages. For GUI elements, the internationalization support translates all possibly affected entries for each translatable component. Thus, changing the language will typically affect the label, tool tip text and hotkey of a button or menu item, and can also impact the optional icon.

The representation of object state presents an important consideration for extensibility. Typically, object state is represented by attributes. The advantages of this approach are well-known: fast access and type checking. However, adding new functionality to the system may in some cases also require the introduction of one or more new attributes, and thus the modification of existing code. In general, we want to avoid the need for code modifications, as any modification of code may introduce bugs and make reading the code more difficult for the next developer.

Therefore, we have decided to represent the object state by properties. In Java, properties are instances of the *java.util.Properties* class, a specialized hash table that can only store String values. The framework also benefits from the default values supported by properties. We only had to implement a customized conversion class that converts various typical attribute types to and from a String representation, for example colors, points or font information. Using properties allows the developer to introduce new property values by simply inserting them into the properties hash at run-time, without requiring any code modification. The runtime differences between property access and direct attribute access are minimal, and are more than made up for by the additional expressiveness we gain.

To make the framework as dynamic as possible, we have further designed to load the individual components dynamically based on a set of configuration files. These files specify the actual components to use, and are located based on the directory where the system was started. Therefore, users can in principle have as many different configurations as there are directories in the file system. Note that this also means that multiple different configurations can share the same code base, for example an end-user and a developer who is currently testing experimental features. In this case, the end-user will be completely unaware of the experimental code.

We maintain the loaded components in a hash structure. The hash table entries act as prototypes for new instances and are cloned to generate the actual components. The framework is therefore able to add or remove individual components at run-time. Note that the removable components are restricted to "non-core" components to ensure that a meaningful system remains after reconfiguration.

Finally, to enhance the development of extensions, we have followed a strict component decoupling. Instead of letting two connected components communicate directly with each other, we introduce a *handler* class as an intermediate agent. The handler is responsible for satisfying queries regarding the offered functionality of a given entity, also based on its current state. When a given functionality is requested for performance, the

handler maps the functionality into the appropriate set of operations. As handlers are tied to classes rather than instances, implementing these operations is usually easy. Additionally, the developer can add or remove handler extensions at run-time, which for all purposes act as if their code were specified inside the original handler code. However, they make touching the handler code for new functionality unnecessary. The strict separation of concerns resulting from this approach makes implementing both components much easier.

4. FRAMEWORK COMPONENTS

The framework we propose consists of five main segments: an import and export layer, a display and editing front-end, and the internal animation representation. Figure 1 shows the conceptual components of the framework. Note that the editing front-end is system-specific and therefore only prepared in the framework, but not directly provided.

The main participating parties in an animation are graphical *primitives*, *animation effects*, and *handlers*. *Primitives* present a simple representation of reusable graphical objects, for example texts, arcs, polygons and points. In our framework, primitives are mainly responsible for two aspects: encoding their current state with appropriate methods for modifying the state, and painting themselves using the standard Java *paint(java.awt.Graphics)* method. Typically, the state can be queried or modified using a pair of *getXX / setXX* methods, where *XX* stands for the name of the property or attribute to retrieve or modify.

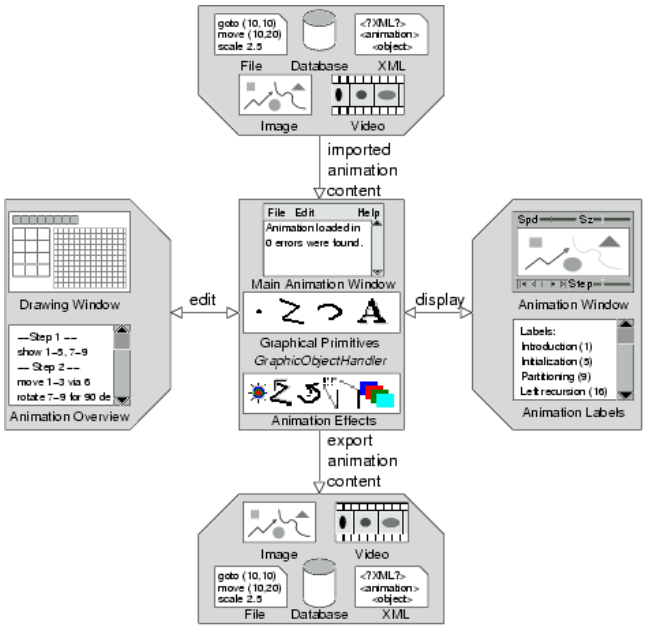

Figure 1. Conceptual Framework Components

Animation effects represent a generic instance of a family of transformation types. For example, all conceivable ways of moving a set of primitives can be encoded in one animation effect, typically called *Move*. The responsibility of the animation effect is twofold. First, it has to determine the concrete animation effect subtype to perform on the selected primitives, for example *"move the second node"*. Second, it has to resolve the timing specification of the effect and determine an appropriate intermediate value.

Handlers in our framework act as negotiating agents between animation effects and primitives. They determine the set of possible transformation types applicable to a primitive and provide this to the animation effect. The animation effect can then build an intersection of all such sets returned by the individual selected primitives and offer this list to the user building the visualization. When a given animation effect has to be performed, the animation effect determines the abstract target state to assume for the current point in time. Abstract target state here means that the state to assume does not depend on the underlying primitives, but only on the timing specification of the animation effect. For example, the target state to assume after 80% of a given move effect have passed is always the same point on the object along which all primitives are moved. The location of the primitives after the operation is highly likely to vary, but their relative movement is still identical. The effect then forwards this information along with the animation effect name to the handler of each selected primitive. The handler is then responsible for mapping the animation effect into a set of method invocations on the primitive that causes the appropriate visual effect.

The handler class offers several key advantages. It offers a very clean separation of concerns for primitives and animation effects: primitives only take care of painting themselves according to their current state, and animation effects only resolve abstract animation operations without having or needing contextual awareness.

Due to the strict separation of concerns encouraged by the handler, developing new primitives or animation effects becomes much easier. When implementing a primitive, the programmer does not have to reflect on how the primitive could be animated. In fact, the implementation can be just the same as for any "static" drawing program! Implementing an animation effect also requires no knowledge about the primitives to animate, but only the skill to interpolate the intermediate states for each point in execution. Finally, implementing a new handler is usually also very easy to accomplish, as the handler is tied to exactly one primitive class type and therefore fully aware of its usage context.

Together, these three classes therefore provide good support for reuse and extensibility. However, the framework goes further than that in two respects. First, the representation of the internal state of primitives and animation effects is not based on fixed attributes, but rather on dynamic lookup structures (java.util.Properties, to be precise). Thus, developers can also add or remove "attributes" for representing the instance at run-time.

Second, the components are not statically assembled based on import statements in the source code. Rather, they are gathered from a configuration file that specifies the names of the classes to load for both primitives and animation effects. Note that this is not possible for handlers, as they are fixed to the primitive they handle. The configured components are loaded dynamically and gathered as Prototypes [2]. Whenever a new instance is required, the prototype is retrieved from the hash table and cloned. Note that the delay caused by this look-up process and cloning is undetectable even when large amounts of objects are used in the system.

The full structure of the animation is separated in two aspects. The "static" view of the animation, consisting of all primitives, animation effects and links connecting the animation steps, is gathered in a Singleton [2] object called *Animation*. The dynamic

state of the animation, for example representing "now" during the display of the animation, is gathered in an *AnimationState* object.

When a given animation state is to be executed, the *AnimationState* clones all animated primitives and causes them to assume the state at the start of the current animation state by quickly executing all previous animation steps. Again, the delay caused by this operation is undetectable. All animation effects are then initialized based on the current point in time. During a looped execution of the animation step, each animation effect checks whether it has to work (that is, the offset before its execution has passed and the effect has not yet finished). If so, the percentage state of execution to assume is calculated from the start time of the animation step, the effect's offset and duration and the current time.

The animation effect determines its target state based on this information. This state is then forwarded to the individual handler Singletons, along with the previous state, the effect name and the current animated primitive. The handler Singleton examines the effect name and both old and new state to determine appropriate method invocations on the primitive. As the net effect, the primitive assumes the correct state as defined by the animation effect.

To allow for better reuse and extensibility, we also support the dynamic integration of *handler extensions* which provide additional animation types for a given primitive. These are also configured and loaded dynamically at run-time and maintained in a hash table.

The import and export layer of the framework provides a simple yet expressive abstract filter class. The individual filters can be added or removed at run-time, just as all other components. The correct filter to use for a file is determined by the user within a specially adapted file dialog. The selection is performed on the basis of the file's MIME type and extension.

The graphical front-end for animation playing supports the setting of the display speed and magnification. It also offers a full-fledged video player control bar, including smooth forward and reverse playing, as well as forward and reverse "slide show" modes. A text field for entering the target animation step to assume and a slider for adjusting the percentage of the animation currently shown are also included.

5. CONCLUSIONS AND FURTHER WORK

The framework as outlined here offers sophisticated support for extensibility and reuse. To prove the flexibility of the framework, we have developed an animation system [3, 4] based on the framework which exhibits the functionality. Developing a system based on the framework presented here is relatively easy, but offers great benefits for both users and animation generators. Note that many key decisions have not been fixed in the framework, such as the type of input needed for generating an animation. Our prototypical system combines three different generation approaches: manual generation within a GUI, generation by scripting and by API invocations. The latter actually generates scripting code.

We hope that other persons interested in AV system development will adopt the framework and thus improve the chances of animation reuse in the field of AV. Of course, much further work is left. We have to formalize the framework specification and provide a careful evaluation of the framework features, both strengths and liabilities. Other interesting approaches for generating animation content, such as code interpretation, can in principle be incorporated into systems built on top of the framework; however, we need to provide a "proof of concept" system that illustrates this. For this purpose, we would be very grateful if other AV system developers could share Java code for parsing and interpreting code, as done for example in the JELIOT 2000 [1] system.

6. REFERENCES

[1] Ben-Bassat Levy, R., Ben-Ari, M., Uronen, P. A. An Extended Experiment with Jeliot 2000. First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press, 2001, pp.131-140.

[2] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[3] Rößling, G., Freisleben, B. Software Visualization Generation Using ANIMALSCRIPT. Informatik / Informatique Special Issues on Visualization of Software, 8:2, April 2001, pp. 35-40.

[4] Rößling, G., Freisleben, B. ANIMAL: A System for Supporting Multiple Roles in Algorithm Animation. Journal of Visual Languages and Computing, 13:2, April 2002 (in print).