IDEA – A Framework for the Fast Creation of Interactive Animations by Pen Sketching

Robert Adelmann
ETH Zurich
Clausiusstrasse 59
8092 Zurich, Switzerland
adelmann@inf.ethz.ch

Tobias Bischoff
University of Freiburg
Georges-Köhler-Allee 051
79110 Freiburg, Germany
tobischoff@gmx.de

Tobias Lauer
University of Freiburg
Georges-Köhler-Allee 051
79110 Freiburg, Germany
lauer@uni-freiburg.de

ABSTRACT

We present a Java-based framework for the easy and on-the-fly creation of algorithm animations. Animations are created by sketching both the objects that should be animated and operations that should be performed on them. The IDEA (Interactive Domain rEcognition and Animation) framework combines the recognition of drawn structures, the interaction with these structures and their animation in order to achieve an effortless and natural creation of animations. The framework itself supports the creation of animations in arbitrary domains through a dynamic plug-in architecture, where so-called domain modules encapsulate all domain-specific semantics. Its application and potential is outlined by the help of three prototypical domain modules for linear lists, Petri nets, and the game Connect-four.

Categories and Subject Descriptors

K.3.1 [**Computer Uses in Education**]: Computer-assisted instruction (CAI); K.3.2 [**Computer and Information Science Education**]: Computer science education—*programming, data structures and algorithms*

General Terms

Algorithms, Human Factors

Keywords

Algorithm animation, sketch recognition, education

1. INTRODUCTION

Research on algorithm animation for educational purposes has identified two major issues to be addressed in order to increase its impact on teaching and learning. The first mainly concerns the students: a growing corpus of empirical studies on the effectiveness of algorithm visualization has revealed that animations should be interactive and engaging in order to be effective [4]. The second issue is a problem chiefly concerning instructors. As a survey among computer science instructors conducted at the *ITiCSE 2002* conference found, the main reasons for many teachers reluctance to use algorithm visualizations in their courses is the time it takes to find good examples, the time to learn how to use a visualization tool, and the time to create or adapt the visualizations [4]. Even though a great number of algorithm animation systems are freely available and offer easy ways of creating visualizations, it is apparently too time-consuming to build useful examples, especially when actual programming is involved. It is therefore essential to support instructors in creating visualizations as quickly and with as little effort as possible with an easy-to-use system.

Visual editors such as provided by systems like [7] are one step in this direction, as they allow a quick, drag-and-drop style creation of animations. The major drawback is that these animations are completely predetermined and do not allow for interaction such as changing the input of the algorithm. Another approach is a generator framework described in [6]: based on a library of existing algorithm animations, instructors can generate customized examples fast and easily by choosing from a variety of options such as input data or the look of the animation. One disadvantage remains for teachers: the use of animations in the classroom is still restricted to pre-constructed examples. If an instructor wants to react to student questions by showing a different example or slightly change the current one, they would have to use the authoring tool in the middle of class.

Our approach to the rapid generation of interactive visualizations aims at its integration with the classroom activity: we present a system that allows a teacher, during his or her presentation, to simply sketch an example of a data structure, say, a binary search tree, on an interactive display. The system automatically recognizes the structure and, upon simple pen gestures by the instructor, carries out the steps of an algorithm, for instance a rotation for rebalancing the tree, in a smooth animation directly on the instructors input drawing. With such a system, teachers are not only able to create animations virtually without effort; they can also respond quickly to comments from students, ask and answer "what if" questions, or let students create and present their own examples. When integrated with other hardware and software in the classroom, such a system also comes close to the goals of *ubiquitous computing*, as the instructor would not even have to explicitly activate the animation system or switch back and forth between it and the normal presentation. The sketches can be drawn directly on the presented slides. If the system is to be used by learners outside class

ITiCSE'07, June 25-27, 2007, Dundee, Scotland, United Kingdom.

on their private computers, they can construct arbitrary examples in a way that is much easier than interacting with most existing animation systems. They can use it to prepare in-class presentations. Moreover, collaborative learning is supported by allowing students to share the drawing panel and construct visualizations together.

2. GENERAL APPROACH

Recognizing structures from pen-based input can be seen as a chain process involving several steps, as is also described in [1] and [8]. First, the input traces must be classified as either a primitive shape or a gesture command. The details of this low-level recognition process are not part of our work, as shape recognition is an extensive field of research of its own. We have incorporated an existing system, which provides robust recognition of primitive shapes and gestures based on a small number of training examples [3]. These primitive objects (or collections of them) and gestures must then be classified as domain-specific objects and commands. For example, a circle together with a number drawn inside it may be classified as a node in the binary tree domain; a "crossing out" gesture drawn on top of a node object may be recognized as a delete command. Finally, interrelations between the domain-specific objects must be detected to infer structures; for instance, two nodes connected by a line should be interpreted as parent and child nodes in a tree. Likewise, commands must be interpreted as invocations of actions performed on the *structure* (rather than just on one single object). For example, deleting a node in a tree usually involves a change of the complete structure, which implies much more than just removing the respective node object from the visualization. In order to do this, an actual instance of the data structure must be available within the system. The presence of such an internal data structure and the possibilities this provides distinguishes IDEA from other sketch-based animation systems like [2].

When an action is carried out, it usually changes the structure and, consequently, its graphical representation. In our binary tree example, the rotation of a node will alter the shape of the tree considerably. Hence, the results of such actions must be propagated back to the original input visualization, where the collection of primitive objects is updated in order to visualize the change. Our contribution and the goal of our system is to combine methods for shape and structure recognition with actual implementations of structures, which are instantiated according to the recognized input. Once the actual structure exists, it can then be manipulated, and the result will be visualized on the drawn input with the help of smooth animations. We note that our main focus is not on sophisticated recognition methods but on an open architecture supporting the complete process chain, in which existing or future recognition and visualization technologies as well as content domains can easily be plugged in.

3. ARCHITECTURE

The IDEA framework can be divided into two parts, the UI component responsible for user interaction and the execution of animations, and the framework which maintains a global system state and manages the different domain modules. The domain modules contain all semantic knowledge for a certain application domain. All other elements, including the UI component, are completely domain independent. Figure 1 shows the basic architecture.

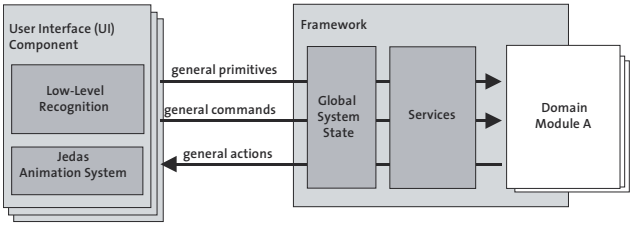

Figure 1: Basic architecture of the IDEA system.

IDEA has been implemented as a distributed system allowing multiple users to work concurrently on the same data. Since both the UI and framework are implemented in Java, the Java RMI technology[1] is used for communication.

3.1 User Interface (UI) Component

The user interface simply consists of a white canvas on which the user can draw sketches. On a standard PC the input device will typically be a mouse but ideally, it is an electronic stylus. According to our experiences, pen input provides a much more natural feedback, especially on interactive boards or on tablet PCs. The sketches drawn by the user are passed on to a low-level recognition system which classifies the pen traces into two classes: primitive graphical objects and gestures [3]. Graphical primitives are lines, polylines, ovals, rectangles, or text. Like the gestures, they are completely domain independent. Gestures include, for example, a stroke connecting two objects, a circle around an object, or a cross. The low-level recognizer currently used relies on training of the primitives and gestures with some examples. If a user wants to draw things that should not be interpreted by the system (such as additional notes), this can be done by holding a button on the pen or pressing the right mouse button while drawing.

The UI component can be seen as a black box that takes pen traces as an input and separates them in two kinds of output: For each graphical primitive the user draws, a so called *general primitive object* is created and sent to the framework. This object contains the basic information about that element, e.g. its position, size, color. Accordingly, for each gesture a *general command object* will be created, which will also be sent to the framework component. In addition, the UI component receives so called *general action objects* from the framework containing animation directions. E.g. "move general primitive object o to position xy". General objects have been introduced in order to allow the remaining system components to abstract from the concrete implementation of the UI component.

For displaying and animating the contents, the UI component uses JEDAS[2], a freely available Java-based library for animation production, which allows the easy creation of sophisticated 2-d keyframe animations. Figure 2 shows a screenshot of the UI component. The tool palette on the left is optional and can be used to select different colors or to enter text using a keyboard.

[1]See `http://java.sun.com`
[2]See `http://ad.informatik.uni-freiburg.de/jedas/`

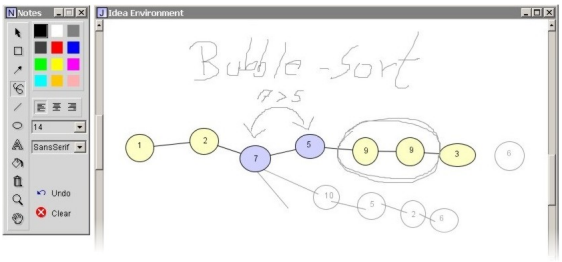

Figure 2: Screenshot of the user interface component with an optional tool-window.

3.2 Framework

The framework works as a server to which several UI components can be connected at a time. It maintains the global system state, which consists mainly of a list of *general primitive objects*. The objects contained in this list are visible on all connected UI components, while the framework ensures the synchronicity of the connected components with this list.

The second task of the framework component is the provision of several domain independent services. These services include the following ones:

- *Recording and Playback* The framework component provides the possibility to record and play back arbitrary situations. Recording and playback of situations can be contolled by gestures. In a teaching scenario, this feature allows for the identification of critical passages and their replay once questions from the audience arise regarding a certain point or when things should be repeated for clarity reasons.

- *Rights Management* The rights management prevents the connection of unregistered UI components to the framework and allows us to assign different rights to the connected UI components. In a typical classroom scenario, for example, it is possible to grant all rights to the teacher and to restrict the other connected UI components belonging to students to merely display what is happening, without the possibility to interfere. One could also restrict the receiving of *general primitive objects* from students but allow *general command objects*, in which case they would be able to work with the structures the teacher has created but would not be able to draw objects on their own.

- *Higher-level Recognition System* The recognition system consists of a collection of tools that support the domain modules in the recognition and interpretation of the primitive objects. It provides, for example, the functionality to determine which graphical primitives are contained in others or which primitives fulfill certain conditions regarding their attributes like size, color, or more complex properties.

Besides the previously mentioned two functions, the framework component manages the domain modules currently plugged in. Different domain modules can be active at the same time. If a module registered at the framework is active, the framework will forward the received *general primitive objects* and *general command objects* from the connected

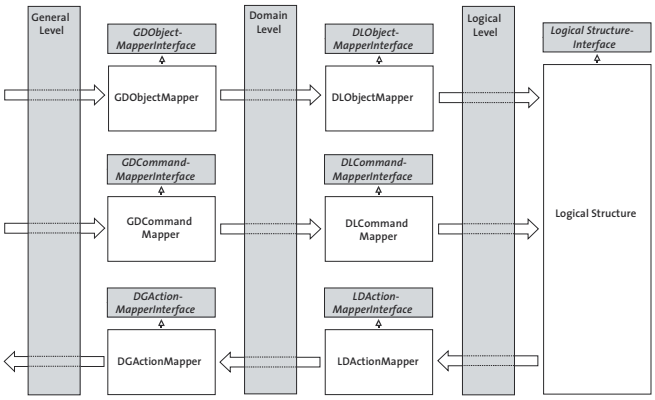

Figure 3: Basic structure of a module. The white parts contain domain specific knowledge, while the grey parts are provided by the framework.

UI components to that module and pass all *general action objects* that are produced by a module on to the UI components.

3.3 Modules

As already mentioned in the last sentence, domain modules can also be seen as black boxes performing some kind of transformation. They receive *general primitive objects* and *general command objects* as input and produce *general action objects* as output. This transformation is done according to the semantics of the domain represented by the module. Even though *general action objects* represent the only means a module can influence the global system state and therefore the display on the connected UI components, modules are in no way restricted in their capabilities. They are normal software components that can use the full power of a programming language.

Internally, modules can be structured in an arbitrary way, but the matrix like structure presented in figure 3 proved to be very effective. Horizontally there are the three information streams: from left to right the primitive or object stream, below it the command stream and a stream of action objects from right to left. Vertically we see three semantic levels: on the left the general level, in which there are general objects that contain no semantic knowledge. In the middle there is a so-called domain level, in which domain-specific objects are available; on the right there is the logical level, in which we have the pure implementation of the data structure and algorithms the module represents. These levels are similar to the three semantic levels used in [1] and [8] for the recognition of structures.

A domain module that can be plugged into the framework basically consist of the white components visible in figure 3. Writing a domain module corresponds to the implementation of these white components.

This structure supports the basic recognition and animation creation processes described in Section 2. How these tasks are performed along this structure can be illustrated using a simple example from the linear-list domain module that encapsulates a singly-linked linear list: We will start with the object stream. Once a new general primitive like a circle or a number has been drawn, the framework will

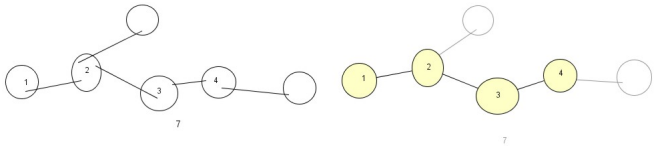

Figure 4: Snapshot of the linear list module. On the left a situation with a low feedback level, on the right a similar situation with a higher feedback level.

pass this information on to the *GDObjectMapper*(General-Domain Object Mapper) component. This component constructs domain objects out of the available general primitives. For example, if it detects that a number has been drawn inside a circle, it constructs a "node" domain object. The linear list module has two types of possible domain objects: nodes and links. The domain objects will be passed on to the *DLObjectMapper*(Domain-Logic Object Mapper) component, which will adjust the implementation of the linear list contained in the *logical structure* component based on the domain objects already recognized and the relations between them.

The processes along the command stream are similar. The module receives general commands like "cross gesture performed at position *xy*". The *GDCommandMapper* component will use this information and construct domain commands like "delete object at position *xy*". Afterwards, the *DLCommandMapper* component detects the domain object present at position *xy*. It uses this information in combination with the current logical structure to create a logic command like "delete node *n*". This logic command will then be passed on to the logic representation of the linear list contained in the *logical structure* component, where it will be executed.

At the action stream, the direction is reversed. The *logical structure* component creates and issues a logic action like "remove node *n*" and passes it on to the *LDActionMapper* component. This component creates a "fade out domain node" domain action, a "fade out link" for the following link and "adjust link" domain action for the preceding link, which will be passed on to the *DGActionMapper* component. There, the *general primtive objects* associated with the specified domain objects are detected (each domain object contains a list of *general primitive objects*) and respective *general action objects* for these graphical primitives are created. The framework will pass these *general action objects* on to the connected UI components.

4. IMPLEMENTED MODULES

Due to space constraints only three of the four domain modules implemented so far are presented. For each module, the domain it models and the basic options for interaction are presented. The binary tree module has been omitted, as it is similar to that for linear lists.

All modules offer different "feedback levels". The feedback level determines the level of "beautification" of the elements drawn by the user once the module recognized parts of the structure, in order to increase the overall clarity of the drawn structures. Figure 4 illustrates the difference.

4.1 Linear List Module

The linear list module has already been mentioned in section 3.3. It encapsulates a singly-linked linear list as well as several sorting algorithms. This module allows the user to draw a linear list and modify it in various ways. The linear list consists of nodes and links. Ovals including text objects are recognized as nodes and lines connecting two nodes are recognized as links. All other drawn elements are treated as comments. For the recognition of the linear list, it is irrelevant in which fashion or order these elements are drawn. Figure 2 shows a typical situation.

Once nodes and links have been drawn, several ways of interaction with the structure are possible. Users can switch the positions of two nodes, mark a selected set of nodes or insert new nodes between two previously marked nodes. If a node or link gets deleted, the resulting linear list will be recognized accordingly. Furthermore, arbitrary sorting algorithms can be executed on the drawn list by performing certain gestures, either step by step, or in one animation, until the whole list is sorted. All actions resulting from these interactions are smoothly animated.

An obvious application scenario of this module could be the discussion of different sorting algorithms during a lecture (in this case, the list may be interpreted as an array). Arbitrary situations can be drawn, animated and augmented with annotations. Due to the interactivity of the system, instructors can react spontaneously to any questions of students, which we think is a great advantage over existing animation systems.

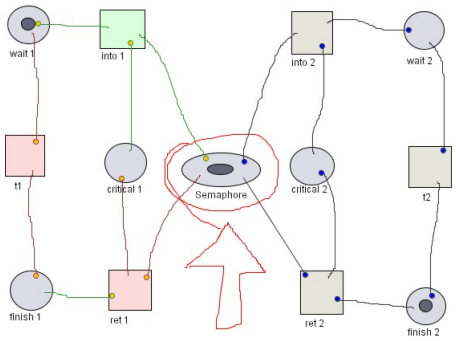

Figure 5: Snapshot of the Petri net module. The three transition nodes on the left side are marked.

4.2 Petri Net Module

This module represents and simulates a condition/event system, a subclass of Petri nets [5]. Petri nets were chosen because they have a more complex structure than linear lists. The module allows the user to model a net by drawing places (state nodes), transition nodes and tokens and by connecting the nodes using directed links. Figure 5 shows an example of a sketched net that models a mutual exclusion situation, in which two processes cannot be both in a critical situation at the same time.

Using certain gestures the firing of only one selected transition nodes, or all nodes at once can be simulated. Users can select all transition nodes inside a certain area using gestures. Selected transition nodes signal, through their color and the color of links connected to them, whether or not they are enabled and why this is so. In the example shown in Figure 5 all three transition nodes on the left side have been marked. It can be seen that the node labeled *ret1* is

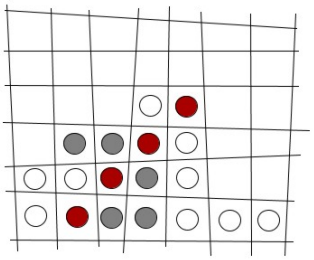

Figure 6: Snapshot of the Connect-four game module using a high feedback level.

not enabled as it is highlighted in a different color. The reason for that can be seen at the color of the links connected to it. The link to the *critical1* place is red, because that node is a preceding node of *ret1* and contains no token. The link to the *semaphore* place is also red, because it is a successor node of *ret1* and already contains a token.

This module would also be well suited for an application in teaching. Furthermore, it can be used for the fast and easy collaborative design and testing of simple condition/event systems. The possibility to clearly visualize a model's status with the mentioned coloring of transition nodes fosters the understanding of the underlying processes in both areas.

4.3 Connect-Four Game Module

This module encapsulates the game Connect-Four[3]. While the previous modules contain more formal structures, this module illustrates the flexibilty of our framework by modeling a more complex system that contains, besides the datastructure itself, quite extensive additional elements like a game engine. It demonstrates a big advantage of the matrix structure presented in Figure 3: The fact that once the object, command and action mapper components that recognize and animate the basic elemements like the playfield, stones or moves, are implemented, arbitrary complex algorithms and systems can be easily built on top of the *logical structure*, without having to consider the underlying recognition or animation processes. In this case a game engine.

Users can start the game by either selecting an area in which a playfield will be created automatically, or they create the playfield manually by simply drawing it. No special care has to be taken regarding too short lines or to the right distance and orientation of the lines. The module will detect and remove all non valid lines automatically. A turn can be executed by drawing a game token and by performing a stroke gesture from that token to the column in which it should be inserted. The resulting animation moves the token to the top of the selected column and drops it into that column. Another gesture triggers a move of the computer player that will automatically select a suitable column according to a simple heuristic. Except for inserting tokens, users have several other ways of interaction: They can clear all tokens inside a playfield, completely remove playfields or create and use multiple playfields at a time, as well as delete selected tokens contained in a playfield. The latter will result in a situation where all tokens above the deleted one will fall down by one field. If one player wins, the set of tokens responsible for the winning will be highlighted.

[3]See `www.wikipedia.org/wiki/Connect_Four`

As an interesting side note, due to the underlying distributed structure of our system, we get a multi-user network game "for free" if multiple UI components are connected.

5. CONCLUSIONS

Motivated by the goal of increasing user engagement and facilitating the creation of data structure animations, we have introduced a system which enables users to simply sketch the structures that should be animated. The IDEA framework supports the combination of the recognition and animation creation process, based on a logical representation of the specific structure. In order to achieve a very open system and to enable it to be successfully applied in a collaborative teaching environment, it has been implemented as a distributed client-server system.

Since the framework itself is domain-independent, it is applicable to a wide range of different application areas. The domain-specific knowledge is encapsulated in modules. These modules work as plug-in components for the IDEA framework. There are two main ideas behind the framework's support for the recognition and animation creation processes performed inside a module: The separation of the processes into object, command, and action streams and the introduction of three semantic levels: the general, domain, and logical level. The result is the matrix structure presented in figure 3. The functioning of this system has been detailed by a series of example modules. So far, the usability of the framework has only been assessed informally in some individual presentations. Future work includes the integration of the system in a classroom environment and its use in actual teaching scenarios.

6. REFERENCES

[1] C. Alvarado and R. Davis. Sketchread: a multi-domain sketch recognition engine. In *Proceedings of UIST 04*, Santa Fe, NM, October 2004.

[2] R. C. Davis and J. A. Landay. Informal animation sketching: Requirements and design. In *Proceedings of the AAAI 2004 Fall Symposium on Making Pen-Based Interaction Intelligent and Natural*, Arlington, VA, October 2004.

[3] K. A. Mohamed and T. Ottmann. Fast interpretation of pen gestures with competent agents. In *Proceedings of CIRAS 2003*. Singapore, December 2003.

[4] T. Naps, G. Rössling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. Velazquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *ACM SIGCSE Bulletin*, 35(2), June 2003.

[5] P.S. Thiagarajan. Condition/Event Systems. Lecture notes, Aarhus University, Mat. Inst., October 1982.

[6] G. Rössling, T. Ackermann, and S. Kulessa. Visualisierung von Algorithmen und Datenstrukturen. In *Proceedings of DeLFI 2006*. Darmstadt, Germany, September 2006.

[7] G. Rössling, M. Schüler, and B. Freisleben. The ANIMAL algorithm animation tool. In *Proceedings of ITiCSE 2000*, Helsinki, Finland, July 2000.

[8] L. Wenyin. On-line graphics recognition: state of the art. In *Proceedings of GREC 2003*. Springer, 2004.