CHAPTER 7

PROTOTYPE LANGUAGE AND SYSTEM

[One of computer science's] distinctive activit[ies] is building things, specifically computers and computer programs. Building things, like fieldwork and meditation and design, is a way of knowing that cannot be reduced to the reading and writing of books. To the contrary, it is an enterprise grounded in a routine daily practice. Sitting in the lab and working on...[computer] programs, it is an inescapable fact that some things can be built and others cannot. (Agre, 1997, p. 10)

The four hypotheses posited in Chapter 5 have important implications for the design of AV technology to be used as a pedagogical aid within the scope of an undergraduate algorithms course. Table 11 summarizes the specific design implications of each hypothesis.

At first glance, an AV system built according to the design guidelines listed in Table 11 would differ in at least three fundamental ways from extant AV systems. First, whereas extant systems almost universally support the creation of input general, high typeset fidelity AVs, the guidelines recommend supporting the construction of low fidelity, input specific AVs. Second, in order to create such AVs, users of many extant systems are required to use quantitative, rather than direct, graphics. Finally, whereas extant AV systems almost universally support Brown’s (1988) “playback” (start, stop, pause) interface for controlling AV execution (see Chapter 2), and almost universally prohibit users from marking up or modifying an AV as it is executing, the guidelines advocate both flexible (i.e., forwards and backwards) execution control and support for dynamic mark-up and modification.

While these guidelines make specific recommendations regarding what tasks users should be able to accomplish with the system (viz., the creation and presentation of low epistemic fidelity AVs), they offer relatively little guidance with respect to how users should accomplish those tasks. For example, the guideline that AV creation should be via direct manipulation constrains the design space, but stops short of specifying the precise user actions for accomplishing that task. In addition, notice that these guidelines fail to furnish the vitally important conceptual model for the system— that is, the underlying model that assists the user in understanding how to use the system. Thus, while these guidelines would appear to serve as a helpful starting point for the development of an alternative AV system—one that supports both the creation and presentation of low epistemic fidelity AVs—they stop short of furnishing a detailed design specification for the system.

In this chapter, I explore the design space circumscribed by the guidelines more fully by presenting an actual prototype AV system grounded in them. In so doing, I aim not only to demonstrate that a system rooted in the hypotheses is practical and feasible, but also to illustrate, as concretely as possible, what such a system might look like. The hope is that the prototype system will draw out the vivid contrasts between the design implications of EF Theory and the sociocultural constructivist-inspired theory being developed in this dissertation.

The foundation of the prototype is SALSA (Spatial Algorithmic Language for Storyboarding), an interpreted, high-level language for programming low epistemic fidelity AVs. Whereas conventional AV technology requires one to program a high epistemic fidelity AV by specifying explicit mappings between an underlying algorithm and the AV, SALSA enables one to specify a low epistemic fidelity AV that drives itself; the notion of an external “driver” algorithm is jettisoned altogether. In order to
support AVs that drive themselves, SALSA enables the logic of an animation to be specified explicitly in terms of the spatiality of the AV—that is, in terms of the spatial relations (e.g., above, right-of, in) among objects in the AV.

Table 11. Implications of Framework Hypotheses for AV Technology Design

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Implied Guidelines For AV Technology Design</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>The Activity Relevance Hypothesis</em>: Low input generality, low typeset fidelity, and direct graphics cause high activity relevance</td>
<td>1. Support the creation of low typeset fidelity, one-shot AVs via direct manipulation, in order to promote engagement in relevant activities</td>
</tr>
<tr>
<td><em>The Communication Effectiveness Hypothesis</em>: Low epistemic fidelity causes high communication effectiveness</td>
<td>2. Support the creation of low typeset fidelity AVs, in order to promote high communication effectiveness</td>
</tr>
<tr>
<td><em>The Understanding and Recall Hypothesis</em>: Self-constructing AVs with a story line, and then presenting them to an instructor for feedback and discussion, causes high recall and understanding</td>
<td>3. Support student AV construction, in order to promote students' active construction of their own understanding</td>
</tr>
<tr>
<td></td>
<td>4. Support construction of AVs with a story line, in order to promote recall</td>
</tr>
<tr>
<td></td>
<td>5. Support easy AV mark-up and modification, as well as flexible control of AV execution, in order to support communication about algorithms</td>
</tr>
<tr>
<td><em>The Community-Building Hypothesis</em>: Self-construction and high instructor communication cause high community-building</td>
<td>6. Use AV technology as the basis for pedagogical exercises in which students construct their own AVs, and then present those AVs to their instructor for discussion, in order to promote increasingly expert participation in the community</td>
</tr>
</tbody>
</table>

The second key component of the prototype is ALVIS (ALgorithm VIualization Storyboarder), an interactive, direct manipulation front-end interface for programming in SALSA. ALVIS strives to make constructing a SALSA animation (i.e., a “storyboard”) as easy as, and ideally easier than, constructing a homemade animation out of simple art supplies. To do so, its conceptual model is firmly rooted in the physical metaphor of art supply storyboard construction, which, in previous empirical studies, has proved to be exceptionally natural, quick, and easy. Moreover, in contrast to existing AV technology, ALVIS also aims to support the interactive presentation of storyboards. To do so, its presentation interface goes beyond existing AV technology’s standard “playback” (start, pause, step) interface by supporting forwards and backwards stepping and execution, as well as dynamic mark up and modification.

In the remainder of this chapter, I present the design and implementation of ALVIS and SALSA in greater detail. I begin by deriving a specific set of design requirements for the prototype from actual empirical studies of humans constructing and presenting homemade animations. Next, I briefly describe the language and system by dissection. To make the descriptions more concrete, I then present an example in which the language and system are used to construct and present an actual (i.e., one observed in empirical studies) homemade animation of the insertion sort algorithm. Finally, having presented the prototype, I discuss its relationship to extant AV systems, as well as its implementation.

### 7.1 Empirical Foundations

In surveying the literature on AV technology, one cannot help but notice a pattern: namely, designers’ intuitions have most often been the guiding force behind its design. One might call this
practice “armchair design.” An important goal of the system-building effort presented in this chapter is to move beyond armchair design by grounding the design firmly in empirical data as part of an iterative, user-centered design process (see, e.g., Norman & Draper, 1986). In this section, I make explicit the ways in which empirical data has informed the design of SALSA and ALVIS. Specifically, I present a corpus of relevant empirical data, from which I derive a set of high-level design requirements.

The design requirements are based on three main sources of empirical data:

1. the two ethnographic studies presented in brief in Chapter 4, and described more fully in Appendices A and B;
2. past empirical studies of how humans construct algorithm animations from conventional art supplies (Chaabouni, 1996; Douglas, Hundhausen, & McKeown, 1995; 1996); and
3. an unpublished pilot study I conducted in which computer science students were asked to watch algorithm animations and then to describe them.

The pertinent observations from these empirical studies give insight not only into what the homemade AVs created by users of ALVIS and SALSA should contain, but also into how users should construct, execute, and present their homemade AVs. Below, I briefly summarize the key observations with respect to AV content, AV construction, and AV execution and presentation.

7.1.1 AV Content

In my ethnographic fieldwork, I gathered well over 100 examples of AVs constructed by 300-level algorithms students. Given the finding that low epistemic fidelity AVs better support the objectives of a 300-level algorithms course than do high epistemic fidelity AVs, the forty storyboards that were constructed using conventional art supplies (including transparencies, pens, and paper) provide the most relevant design information. These storyboards depicted divide-and-conquer, greedy, dynamic programming, and graph algorithms, including Huffman compression, QuickSelect, breadth-first and depth-first search, and Dijkstra’s shortest path algorithm. The following generalizations can be made regarding their content:

1. The storyboards consisted of groups of movable, labeled objects of arbitrary shape (but most often boxes, circles, and lines) arranged on a static background; regions of the display, and locations in the display, were also significant.

2. Objects in storyboards were frequently arranged according to one of three general layout disciplines: grids, trees, and graphs.

3. The most common kind of animation was simple movement from one point to another; pointing (with fingers) and highlighting (circling or changing color) were also common. Occasionally, multiple objects were animated concurrently—for example, two objects moving concurrently along the same path.

In a sense, empirically-driven design is what this dissertation is all about. Indeed, the theoretical shift for which I am arguing emerged not by sitting in an armchair and imagining how AV technology might be pedagogically effective, but rather by using empirical data as a basis for understanding what algorithms students and instructors actually do with AV technology, and how it contributes to their understanding, identity, and competence.
4. The majority of the storyboards unfolded within a single “window” (rectangular area); occasionally, multiple “windows” depicted multiple, synchronized views of an algorithm, or multiple views of alternative algorithms operating on the same data set (e.g., breadth-first vs. depth-first search).

7.1.2 Process of AV Construction

Douglas, Hundhausen, and McKeown (1995; 1996) and Chaabouni (1996) used Interaction Analysis (Jordan & Henderson, 1995) to study in detail the process by which humans construct AVs using simple art supplies. Their observations suggest that AV storyboard designers create objects by simply sketching them out (or cutting them out) and placing them on the page. Design tends to be an experimental, dialectical process, rather than a calculated, linear one. For example, new objects are added to the storyboard as the need for them arises. Similarly, decisions regarding how to animate these objects tend to be made tentatively on the spot, not firmly in advance. Furthermore, the size of the objects, as well as their placement, is invariably done not in terms of absolute coordinates, but rather relative to other objects that have already been placed in a storyboard; AV designers never place or move objects according to Cartesian coordinates.

In an unpublished pilot study in which I asked students to view and describe algorithm animations, I made three observations that complement the ones just presented. First, participants invariably gave animation objects descriptive names; however, two participants rarely chose the same name for the same object. Second, participants tended not to refer to animation objects using Cartesian coordinates. Instead, they referenced objects by using either descriptive labels, or by describing them in terms of their spatial relationship to other objects, which served as landmarks. Finally, participants tended not to supply exact destination locations of animation actions (e.g., move object1 to 20,30). Rather, destination locations were specified either (a) as an implicit location described in terms of its spatial relationship to other objects (e.g., move object1 below object2 or move object1 below object right-of object1); or (b) as an implicit location described in terms of a direction with respect to an object’s present location (e.g., move object1 left). In the case of (b), the distance that the object was to be moved often was not explicitly specified, but was usually the length or width (or fraction thereof) of an adjacent object.

7.1.2 Process of AV Execution and Presentation

In this process, AV designers set their storyboard into motion to illustrate the execution of an algorithm on a particular data set. Observations made by Douglas, Hundhausen, and McKeown (1995, 1996) and Chaabouni (1996) indicate that, rather than referring to program source code or pseudocode, AV storyboard designers tend to simulate their storyboards by paying close attention to, and hence maintaining, important spatial relations among storyboard objects. For example, rather than maintaining a looping variable and stopping a loop when that variable reaches a certain value, storyboard designers might instead observe whether an object representing that loop variable reaches a certain position in the storyboard. For example, they might stop looping when an object advances to the right of a row of boxes.

In the presentation sessions I observed in my fieldwork, students provided verbal play-by-play narration as they executed their storyboards. They used deictic gestures (both with their fingers and the tips of pens) extensively to coordinate their narration with objects in their storyboards. Frequently, audience members broke in with comments or questions. To clarify such comments and questions, audience members often pointed to objects in the storyboard, or even marked up the storyboard with a pen. In response to such comments and questions, student presenters paused their animation, or even fast-forwarded or rewound it to another point of interest. In some cases, audience suggestions led to on-the-spot modifications of the storyboard—for example, changing a color scheme, adding a label, or altering a set of input data.
7.1.3 Design Requirements

Taking the above observations into account implies five high level functional requirements for SALSA and ALVIS:

1. **Storyboard content.** SALSA and ALVIS must be capable of expressing the AV storyboards that I observed in my ethnographic fieldwork.

2. **Storyboard creation process.** SALSA and ALVIS must enable users to express storyboards using the same dialectical, experimental process that empirical study participants adopted. In other words, the system encourage experimentation by supporting a short modify-compile-execute cycle.

3. **No Cartesian coordinates.** SALSA and ALVIS must enable users to express storyboards in the same terms that the study participants used—using spatial relations, not Cartesian coordinates.

4. **Spatial execution model.** SALSA and ALVIS must support an execution model similar to that adopted by empirical study participants—one based on spatial, rather than algorithmic logic.

5. **Interactive presentation.** SALSA and ALVIS must enable users to present their storyboards as part of interactive discussions. This entails an ability to pause and restart storyboards; to rewind and fast forward storyboards to points of interest; and to point to, mark-up, and modify storyboards as they are being presented.

In addition, two high level usability requirements are implied by the empirical data:

1. **Quick and Easy Storyboard Creation.** Undergraduate algorithms students should be able to create homemade AVs quickly and easily with SALSA and ALVIS—at least as quickly and easily as they can create storyboards using conventional art supplies, and ideally more quickly and easily.

2. **Fluid, Interactive Presentation.** Undergraduate algorithms students should be able to use SALSA and ALVIS to fluidly present their animations to an audience. In other words, in response to audience questions and comments, student presenters should be able to control their animations’ execution easily, and to modify and mark up their animations easily; the interface should not get in the way of the presentation.

In the following two sections, I give brief tours of the two key components of the prototype technology designed to meet these requirements: SALSA and ALVIS.

7.2 SALSA

The foundation of the prototype system is SALSA, a high-level, interpreted language for programming low epistemic fidelity AVs, or storyboards. In line with the requirements identified above, SALSA scripts (a) produce low typeset fidelity storyboards (i.e., they resemble sketches,
rather than textbook figures); (b) are devoid of Cartesian coordinates; are capable of expressing spatial relations and logic; and (c) operate on a specific set of input data, rather than general input.\textsuperscript{51}

The remainder of this section describes SALSA by dissection, beginning with the data types, moving to the language's support for spatial relations, and concluding with a synopsis of the SALSA commands. A more detailed summary of the SALSA language can be found in Appendix F.

### 7.2.1 Data Types

SALSA defines three data types, each of which models one of the core elements of the “art supply” storyboards observed in the empirical studies discussed above:

1. **Cutout.** This is a computer version of a construction paper cutout. It can be thought of as a movable scrap of construction paper of arbitrary shape on which graphics are sketched. Associated with the cutout data type are several attributes. Some of these are purely spatial, and define various x,y positions (called "refpoints") in and around the cutout (e.g., left-center, top, bottom; see Figure 39). Others have to do with the appearance of the object, such as graphic-rep (a file containing the graphics of the cutout), visible, and highlighted.

2. **Position.** The position data type represents an x,y position within a storyboard.

3. **S-struct.** As discussed in the previous section, empirical studies suggest that AV designers use the same spatial layout patterns time and time again. A spatial structure (s-struct for short) can be thought of as a closed spatial region in which a set of (not necessarily homogeneous) cutouts can be systematically arranged according to a particular spatial layout pattern. The prototype implementation of SALSA supports just one s-struct: grids.

![Predefined Cutout Refpoints](image)

Figure 39. Predefined Cutout Refpoints

\textsuperscript{51}This point requires further explanation. A given SALSA script operates on objects that represent one specific set of input data. However, the fact that SALSA is an interpreted language makes it easy to redefine the objects of the script that represent the input data. Hence, as long as the internal logic of a SALSA script is not “hard coded” for a specific set of input data, it is possible to redefine the objects of the script, and then to re-execute the script to observe the animation operating on an alternative set of input data. This kind of “pseudo input generality” will be illustrated in the example presented later in the chapter.
7.2.2 Spatial Relations

As discussed in the previous section, a key requirement is that SALSA programmers not have to deal with Cartesian coordinates in any form. SALSA addresses this requirement by supporting spatial relations, which provide a mechanism both for describing storyboard elements in terms of how they relate spatially to other storyboard elements, and for expressing storyboard logic in terms of the spatiality of a storyboard. Consider, for example, the sample storyboard in Figure 40. The viewer of this storyboard could make a great number of statements about the ways in which the various cutouts in this storyboard are spatially related to each other. For instance, square is left-of hexagon; square is right-of triangle; cross is above square; arrow is below square; diamond is above hexagon; diamond is touching hexagon; oval is in hexagon; and arrow is outside-of square.

![Figure 40. Cutouts in a Sample SALSA Storyboard](image)

One might wonder how diamond can be both above, and touching, hexagon. The answer is that each cutout has a distinct reference point that is implicitly referenced in any spatial relations statement. For the purpose of illustration, each cut-out’s implicit reference point is indicated by a dot in Figure 40. In SALSA, one may set a cut-out’s refpoint attribute to any of the nine different locations shown in Figure 39.

The concept of a view plane adds a third dimension to spatial relations in SALSA. In Figure 40, notice that the diamond is actually closer to us than the hexagon, since it obscures a portion of the hexagon. In SALSA, each cut-out occupies a distinct view plane, making it possible to express spatial relations that involve a cut-out’s “closeness” to us (the viewers)—for example, hexagon is further-away-than diamond, and, conversely, diamond is closer-than hexagon. Finally, notice that Figure 40 includes a position called mypoint. It is also possible, in SALSA, to state and test whether two points are at the same location. With respect to the sample storyboard in Figure 40, one could say top-center of triangle is at mypoint.

In sum, Table 12 lists SALSA’s 10 spatial relations keywords, which enable one to express any number of spatial relations among the elements of a storyboard.

---

52 For an empirically-based account of how humans use spatial relations, see (Douglas, Novick, & Tomlin, 1987).
Table 12. The Ten Spatial Relations Supported By SALSA

- above
- touching
- further-away-than
- below
- in
- left-of
- outside-of
- right-of
- closer-than
- at

7.2.3 Commands

Table 13 briefly summarizes the 12 commands defined by SALSA. As the table indicates, these commands fall into three main categories. The first category of commands supports storyboard element creation, deletion, and attribute modification; the second supports conditional branching and iteration; and the third supports various forms of animation.

Table 13. Brief Summary of SALSA Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>Creates a new cutout, position, or s-struct. The Create as clone variant enables one to create a new object as a copy of an existing object.</td>
</tr>
<tr>
<td>Place</td>
<td>Actually positions a created storyboard element in the storyboard.</td>
</tr>
<tr>
<td>Delete</td>
<td>Deletes a created storyboard element.</td>
</tr>
<tr>
<td>Assign</td>
<td>Creates an alternative identifier for a created storyboard element.</td>
</tr>
<tr>
<td>Set</td>
<td>Sets an attribute of a storyboard element to a new value.</td>
</tr>
<tr>
<td>If-then-else</td>
<td>Supports conditional branching. In most cases, the conditional will test spatial relations of storyboard elements, as described above.</td>
</tr>
<tr>
<td>While</td>
<td>Supports conditional iteration.</td>
</tr>
<tr>
<td>For-each</td>
<td>Supports iteration over (a) a set of positions associated with a grid or cutout, or (b) a set of storyboard elements that satisfy a boolean test.</td>
</tr>
<tr>
<td>Move</td>
<td>Animates a cutout from its present location to a new location over a specified length of time, and along a specified path.</td>
</tr>
<tr>
<td>Resize</td>
<td>Changes the size of a cutout over a specified duration of time.</td>
</tr>
<tr>
<td>Flash</td>
<td>Flashes a cutout over a specified period of time.</td>
</tr>
<tr>
<td>DoConcurrent</td>
<td>Enables a block of animation commands to be executed concurrently.</td>
</tr>
</tbody>
</table>

7.3 ALVIS

As the graphical front-end to the SALSA language, ALVIS provides both a direct-manipulation interface for constructing SALSA storyboards, and an interactive environment for executing and presenting them; it is thus the component of the prototype with which users directly interact. The conceptual model underlying ALVIS's storyboard construction interface is rooted in the physical metaphor of art supply storyboard construction. An important component of this metaphor is the concept of cutouts (or “patches”; see van de Kant, Wilson, Bekker, Johnson, & Johnson, 1998): scraps of virtual construction paper that may be cut out and drawn on, just like real construction paper. ALVIS users create homemade animations by cutting out (with a scissors tool), sketching on (with a pen tool), and arranging (via direct manipulation) cutouts on a static background. They then specify, either by direct manipulation or by directly typing in SALSA commands, the ways in which the cutouts should be animated over time.

Likewise, the conceptual model underlying ALVIS's storyboard presentation interface is based on the physical metaphor of presenting an “art supply” storyboard. The interface supports four distinct
features that are taken for granted in “art supply” presentations, but that are notably absent in conventional AV technology. First, it is possible in ALVIS to reverse the direction of storyboard execution in response to audience questions and comments. Second, ALVIS provides a conspicuous “presentation pointer” with which the presenter and audience members, may point to objects in the storyboard as it is executing. Third, ALVIS includes a “mark up pen” with which the presenter and audience members may dynamically annotate the storyboard as it is executing. Finally, presenters and audience members may dynamically modify a storyboard as it is executing by simply inserting SALSA commands at the current insertion point in the script—that is, the point in the script where execution is paused.

The remainder of this section gives a high level tour of the ALVIS interface. This will set the stage for the section that follows, in which I illustrate, by way of a detailed example, how ALVIS can be used to create and present a storyboard.

7.3.1 Overview

Figure 41 presents a snapshot of the ALVIS environment, which consists of three main regions:

1. **SALSA Script view** (left). This view displays the SALSA script presently being edited; the arrow on the left-hand side of the view denotes the line at which the script is presently halted—the current “insertion point” for editing.

2. **Storyboard view** (upper right). This view displays the (graphical) storyboard generated by the SALSA script displayed in the SALSA Script view. The Storyboard view is always synchronized with the SALSA Script view. In other words, it always reflects the execution of the SALSA script up to the current insertion point marked by the arrow.

3. **Created Objects palette** (lower right). This area contains an icon representing each cutout, position, and grid that has been created thus far. When an icon is double-clicked, a dialog box containing the object’s current attribute values appears. The Created Objects palette thus provides a convenient means of accessing and editing the attributes associated with each object. Note that, like the Storyboard view, this palette is synchronized with the SALSA Script view. In other words, it displays all storyboard objects that have been created as a result of the script executing up to the current insertion point.

Users may access all of ALVIS’s functionality through its menus, which are presented in Figure 42. The File menu contains standard commands for creating, opening, closing, saving, and printing storyboard (.sal) files, which are nothing more than plain text files containing a sequence of SALSA commands. Similarly, the Edit menu comprises standard editing commands, all of which make sense when users are working within the SALSA Script View, but only some of which make sense at other times (e.g., Delete is available when an icon in the Created Objects palette is selected). The View menu toggles the visibility of all non-menu interface elements. The toolbars and Cutout Graphics Editor that appear on the View menu, along with all of the items on the Create, Script, and Present menus, assist users in storyboard creation and presentation; I describe them further in the two subsections on those topics that follow. Finally, the Zoom menu enables the user to set the zoom factor for the storyboard view; the higher the zoom factor, the larger the cutouts appear.

53 Note that, in the prototype implementation, users may work with just one storyboard at a time. Note also that each create cutout command within a SALSA script contains a reference (i.e., a path and file name) to a .cut file containing the cutout’s graphics. (The process by which users create such files will be discussed below.) Thus, in order for the script to run properly, all cutout graphics files referenced in the script must be readable.
7.3.1 Storyboard Construction Interface

In ALVIS, constructing a storyboard amounts to programming a SALSA script. At any point, ALVIS users always have at least two different ways of doing that:

Type SALSA commands directly into the SALSA Script view at the current insertion point.
Use the ALVIS interface to generate SALSA commands by directly manipulating objects in the Storyboard view and/or filling in dialog boxes; any commands so generated are automatically inserted in the SALSA Script view at the current insertion point.\(^{54}\)

The two core subtasks in storyboard creation are (a) creating and placing storyboard elements, and (b) animating cutouts. With the proviso that these tasks may be accomplished by simply typing appropriate SALSA commands into the SALSA Script view, I now briefly discuss how the ALVIS interface supports them.

### 7.3.1.1 Creating and Placing Storyboard Elements

By selecting items on the Create menu and filling in corresponding dialog boxes, users may generate appropriate “create” commands and insert them at the current insertion point in the script. The dialog boxes that appear in response to choosing items on the Create menu present the default values of the attributes for the object (cutout, position, or grid), and invite users to override those defaults.

For example, Figure 43 presents the dialog box that appears in response to selecting Cutout…on the Create menu. Users are invited to name the new cutout, but they need not; upon creation, each SALSA object is automatically assigned a unique ID, which may be used in lieu of a name to reference the object. Users specify a cutout’s graphical by either selecting an existing .cut file (with the “Browse” button), or by creating a new .cut file. The user may create a new graphical representation for a cutout with the Cutout Graphics Editor, which appears in response to clicking the “Create New…” button in Figure 43, and is also accessible at any point through the View menu.

---

\(^{54}\)It is important to note that this is an option only when the user performs edits at the current insertion point. If the user attempts to perform edits at some other point in the script, the direct manipulation editing commands are disabled, since the Storyboard view does not reflect the current state of the SALSA script.
Figure 44 presents a snapshot of the Cutout Graphics editor. As in “art supply” storyboard construction, creating a cutout involves first using the scissors tool to cut out a virtual scrap of paper from a base shape of the user’s choice (square and circle are presently supported), and then using the pen tool to sketch graphics onto the scrap of paper. At any point, pen color may be altered, and sketched graphics may be erased with the eraser tool. Once satisfied, the user may save the graphics to a .cut file; the file is then automatically inserted into the “Source File” field of the Create Cutout dialog box, and is available for use as any other cutout’s graphical representation.

Recall that in SALSA, creating an object and placing an object are two different commands. Accordingly, in ALVIS, an object that is created through one of the Create dialog boxes does not immediately appear in the storyboard view; rather, it must be explicitly placed there. To do this, the user must drag-and-drop the newly created object from the Created Objects palette (where it appears immediately after creation) to a desired position in the Storyboard view; in response, the object appears in the Storyboard view, and ALVIS inserts an appropriate place statement into the script at the current insertion point.

7.3.1.2 Animating Cutouts

Recall that SALSA supports three different forms of animation—move, resize, and flash. In order to specify these forms of animation in ALVIS, users must first select the cutout to be animated in the Storyboard view, and then choose an animation command from the Script menu. In the case of Move and Resize, users are then asked to demonstrate the desired animation by directly manipulating the selected cutout—either by moving it along the desired path to the new location, or by resizing the cutout to the desired size by manipulating the resize handles that appear. In all three cases, a dialog box pops up that invites users to refine the animation further.
For example, Figure 45 presents the *Script Move* dialog box. Using the move demonstrated by the user, ALVIS makes assumptions about the format of the SALSA command that will ultimately be generated, and inserts those assumptions into this dialog box. First, the duration of the demonstrated move is timed, and that time is inserted into the Duration field. Second, ALVIS assumes that the movement should be to an absolute position, and it uses the landing point of the demonstrated move as a basis for filling in the x and y coordinates. Finally, ALVIS assumes a custom path, and generates a series of offsets for that path based on the demonstrated move.

![Figure 45. The Script Move Dialog Box](image)

Clearly, most of these assumptions are naïve. Fortunately, the *Script Move* dialog box enables the user to correct the assumptions. For example, the user may have intended the movement to end at a particular storyboard reference point (e.g., *left-center of foo*). In that case, the user may override the default by clicking on the “Storyboard Reference” radio button and specifying a storyboard reference point in the corresponding text box. Likewise, the user may have actually wanted to drag a perfectly straight path. The user may override the default custom path in that case simply by selecting the “Straight” (Path) radio button.

### 7.3.2 Storyboard Presentation Interface

Because the *SALSA Script* and *Storyboard* views are tightly synchronized, the tasks of storyboard creation and presentation are virtually inseparable in ALVIS. Indeed, in creating storyboard elements and specifying how they should be animated, ALVIS users necessarily execute their storyboard; there is no separate “presentation mode.” In this way, ALVIS supports the dynamic modification requirement outlined above: At any point in a presentation, users may dynamically alter their storyboard simply by inserting or removing commands at the current insertion point.

In addition to dynamic modification, ALVIS supports several features that are specifically designed to assist in storyboard presentation. All of these tools are available on floating toolbars. The *Execution Control* toolbar (see Figure 46), whose functionality is fully duplicated by the items on the
Present menu, provides users with an interface for executing their storyboards. This interface supports both forwards and backwards single-stepping and execution. Moreover, it provides a means of executing to an arbitrary point in the script, as marked by the current location of the cursor in the SALSA Script view.

![Run Toolbar](image)

Figure 46. The Execution Control Toolbar

In discussions up to this point, I have taken it for granted that the first item on the Present/Edit toolbar (see Figure 47), the “selection” tool, is active. During storyboard creation, it enables users to select objects in the storyboard and apply animation commands to them. However, the next four tools on the toolbar are geared toward presentation. When selected, the “presentation pointer” changes the cursor into a conspicuous pointer, while the “mark-up pen” changes the cursor into a virtual marking pen, whose color and thickness may be selected from the palette of colors on the Pen Color toolbar. To erase specific annotations, users may select the “eraser” tool; to wipe all annotations clean, users may select the “erase all” tool.

![Present/Edit Toolbar](image)

Figure 47. The Present/Edit Toolbar

### 7.4 Example of the Prototype in Use

To make the features of SALSA and ALVIS just described more concrete, I now walk through an example in which I use ALVIS to create and present a simple storyboard: the “football” animation of the bubble sort algorithm observed by Douglas, Hundhausen, and McKeown (1995; 1996). Recall that the bubble sort, an $n^2$ algorithm, successively compares adjacent elements, swapping them if they are out of order (see Figure 48). The result is that smaller elements “bubble up” to the front of the array, or, conversely, that heavier elements “sink” to the end of the array. Specifically, after the $n^{th}$ pass of the algorithm’s outer loop (lines 2 to 8), we are guaranteed that the last $n$ elements of the array are in place.

The “football” animation (see Figure 49) depicts the bubble sort algorithm by telling the story of a game of American football. Elements to be sorted are represented as football players whose varying weights represent element magnitudes. At the beginning of the game, the players are lined up in a row. The referee then tosses the ball to the left-most player in the line, who becomes the ball carrier. The object of the ball carrier is to “score” by advancing the ball to the end of the line. If the ball carrier is heavier than the player next in line, then the ball carrier simply tackles the next player in line, thereby switching places with him. If, on the other hand, the ball carrier is lighter than the player next in line, then the ball carrier is stopped in his tracks, fumbling the ball to the next player in line. This process of ball advancement continues until the ball reaches the end of the line, at which point the player at the end of the line tosses the ball back to the referee. A pass of the
algorithm’s outer loop thus completes. If, at this point, there are still players out of order, the referee tosses the ball to the first player in line, and another next pass of the algorithm commences.

```plaintext
1:   BUBBLESORT(A,n)
2:   for j ← n-1 to 1
3:       for i ← 1 to j
4:           if (a[i] > a[i+1])
5:               exchange a[i] ↔ a[i+1]
6:       end if
7:   end for
8: end for
9:   end BUBBLESORT
```

Figure 48. Pseudocode Description of the Bubble Sort Algorithm

![Figure 49. The Football Bubble Sort Storyboard](image)

7.4.1 Creating the Storyboard Elements

Having launched the ALVIS environment, I begin by creating the cutouts that appear in the animation of Figure 49: four players, a football, a referee, and a goal post. With respect to the football players, my strategy is to save time by creating one “prototype” player, and then cloning that player to create the other three players. This will work because I will opt to convey player weight not by varying player height or width, but rather by labeling each player, just as in Figure 49. To create the prototype player, I select Cutout… from the Create menu, which brings up the Create Cutout dialog box. I name the cutout “player1,” and use the Cutout Graphics Editor to create its graphics in the file “player.cut”: a square scrap of construction paper with a football player stick figure sketched on it. In addition, I set the data attribute to “300,” and decide to accept all other default attributes.

When the Create Cutout dialog box is dismissed, ALVIS inserts the following SALSA create statement into the SALSA Script view:

```
create cutout player1 -graphic-rep "player.cut" -data "300"
```

In addition, the “player1” cutout appears in the Created Objects palette, and the execution arrow (in the SALSA Script view) is advanced to the next line, indicating that the create statement has been executed. Figure 50 shows a snapshot of ALVIS at this point.
I now proceed to clone “player1” three times. For each cloning, I first select the “player1” icon in the Created Objects palette, and then choose Clone… from the Create menu. This causes the Create Cutout dialog box to appear, with all attribute settings matching those of “player1.” In each case, I change the name (to “player2”, “player3”, and “player4”, respectively), and I change the data (to “250”, “200”, and “150”, respectively), while leaving all other attributes alone. After dismissing the Create Cutout dialog box each time, a new create statement appears in the script—for example,

```
create cutout player2 as clone of player1 -data "250"
```

In addition, the new cutout appears in the Created Objects palette, and the execution arrow advances to the next line. I proceed to create the football, referee and goal post in the same way.

### 7.4.2 Placing the Storyboard Elements

The next step is to place the cutouts in the storyboard. In order to lay the players out in a row, I use a simple grid s-struct, which I create by choosing Grid… from the Create menu and then filling in the Create Grid dialog box. I name the grid “a,” so that it corresponds with the name of the array in the pseudocode of Figure 48. The grid needs one row and four columns, with a cell height and width corresponding to the height and width of “player.” I accept all other default attributes, and click on the “Create” button to create the grid. Just before it is dismissed, the Create Grid dialog box appears as in Figure 51.

I now proceed to place in the storyboard all of the objects I have created. First, I drag and drop the grid to an acceptable location in the middle of the storyboard; the corresponding place statement appears in the script, with the execution arrow advancing to the next line. With the grid in place, it is straightforward to position the players at the grid points:

```
place player1 at gridpos 1 of a
place player2 at gridpos 2 of a
place player3 at gridpos 3 of a
place player4 at gridpos 4 of a
```
Finally, I drag and drop the referee to a reasonable place below the row of football players, and I drag and drop the football to a place just above the referee. Figure 52 shows the ALVIS environment after all elements have been placed.

Figure 51. Creating a Grid with the Create Grid Dialog Box

Figure 52. The ALVIS Environment After All Cutouts Have Been Placed
7.4.3 Programming the Spatial Logic

I am now set to do the real SALSA programming necessary to make the storyboard work. From this point on, the SALSA code is of more interest than the ALVIS interface for programming it. Hence, I will focus mainly on the code itself in the discussion that follows, mentioning the ALVIS interface only in occasional footnotes.

As each player reaches his rightful place in the line, I want to turn his outline color to red. Since I have set the outline color of the goal post (the right-most cutout in the storyboard) to red, we know that when the outline color of the cutout immediately to the right of the ball carrier is red, the ball carrier has reached the end of the line, and we are done with a pass of the inner loop. In SALSA, we may formulate this as a while loop:

```
while outlinecolor of cutout right-of cutout touching football is red
  --do inner loop of bubblesort (see below)
endwhile
```

This code segment raises an important issue with respect to object resolution in SALSA; it is worth discussing briefly before I move on to code more of the storyboard. Note that the segment `cutout right-of cutout touching football` is potentially ambiguous; there may well be several cutouts that are touching the football, and there may well be several cutouts right of those. To resolve such ambiguous references, SALSA adopts a straightforward approach: Always take the best possible match. Say, for example, that more than one cutout were touching `football`. Then `cutout touching football` would resolve to that cutout which had the most area in common with `football`. Now assume that `cutout touching football` resolves to `player1`, and suppose that more than one cutout were to the right of `player1`. In that case, `cutout right-of player1` would resolve to the cutout that (a) is right of `player1` and (b) is the shortest Euclidean distance from `player1`. In sum, although SALSA supports ambiguous object references, it is important to keep in mind that ambiguous references like the one above may not always resolve to a unique cutout. In fact, in some cases, they may not resolve to any cutout at all.

Getting back to the football storyboard, we know that we are done with all passes of the outer loop when all but the first player in line has a red outline color. (There is no need to compare the first player to the second player, if the second player is known to be in order.) In SALSA, we may express this logic as another while loop:

```
while outlinecolor of cutout at position 2 of a is not red
  --do outer loop of bubblesort
endwhile
```

Now we come to the trickiest part of the script: the logic of the inner loop. We want to successively compare the ball carrier to the player to his immediate right. If these two players are out of order, we want to swap them, with the ball carrier maintaining the ball; otherwise, we want the ball carrier to fumble the ball to the player to his immediate right.

---

55To program a While loop through ALVIS, I could choose While… from the Script menu, which causes a while loop template to be inserted at the current insertion point of the script. Obviously, ALVIS is unable to program the loop for me!
The following code makes the comparison, either fumbling or swapping, depending on the outcome:

```plaintext
if data of cutout touching football > data of cutout right-of cutout touching football
    --swap players
    assign p1 to position in a of cutout touching football
    assign p2 to position in a of cutout right-of cutout touching football
    doconcurrent
        move cutout touching football to p2
        move cutout right-of cutout touching football to p1
        move football right cellwidth of a
    enddoconcurrent
else --fumble ball to next player in line
    move football right cellwidth of a
endif
```

With the inner loop complete, all that remains is to piece together the outer loop. At the beginning of the outer loop, we want to toss the ball to the first player in line. We then want to proceed with the inner loop, at the end of which we first set the outline of the player with the ball to red (signifying that he has reached his rightful place), and then toss the ball back to the referee. Thus, the outer loop appears as follows:

```plaintext
move football to left-center of cutout at position 1 of a
//inner loop goes here
set outlinecolor of cutout touching football to red
move football to top-center of referee
```

Just one small detail remains: When the outer loop falls through, we might want to turn the outline of the first player in line to red, signifying that we are done sorting:

```plaintext
set outlinecolor of cutout at position 1 of a to red
```

---

56To insert an if-then-else template at the current insertion point, I could choose If-then-else from the Script menu. I could specify the attach statement by choosing Attach... from the Script menu, and then clicking, in turn, on the dependent and independent cutouts in the attachment. I could specify the assign statements by choosing Assign... from the Script menu, and then filling in the Assign dialog box. To program a DoConcurrent statement through ALVIS, I could choose DoConcurrent... from the Script menu, which inserts a DoConcurrent template at the current script insertion point. In ALVIS, I can program the flashes by first selecting, one at a time, the cutouts to be flashed, and then selecting Flash... from the Script menu and filling in the Flash Cutout dialog box that comes up.

57What if we wanted to generating this set command through ALVIS? It turns out that this raises an interesting issue. Let us assume that the player presently with the football is “player2.” We could double-click on “player2” in the Storyboard view, which would bring up an Edit Cutout dialog box that resembles the Create Cutout dialog box presented earlier. We could then select “red” from the “Outline Color” drop-down menu and dismiss the dialog box. ALVIS would then generate the following command: set outlinecolor of player2 to red. Notice that what ALVIS generates is more specific than what I have written above—an observation that underscores the fact that ALVIS is too naïve to generate general-purpose code from a specific instance.
Figure 53 summarizes this example by presenting the complete script.

```
1: --create the storyboard elements
2: create cutout player1 -graphicrep "cutout.cut" -data "300"
3: create cutout player2 as clone of player1 -data "250"
4: create cutout player3 as clone of player1 -data "200"
5: create cutout player2 as clone of player1 -data "150"
6: create cutout referee -graphicrep "referee.cut" -data "referee"
7: create cutout football -graphicrep "football.cut"
8: create cutout goal -graphicrep "goal.cut" -outlinecolor red
9: create grid a -rows 1 columns 4 -cellwidth width of player1
10: --cellheight height of player1
11: place a at 0.5,0.5
12: place player1 at gridpos 1 of a
13: place player2 at gridpos 2 of a
14: place player3 at gridpos 3 of a
15: place player4 at gridpos 4 of a
16: place referee at
17: place football at top-center of referee
18: --ready to perform bubble sort. . .
19: while outlinecolor of cutout at position 2 of a is not red --do outer loop
20:   move football to left-center of cutout at gridpos 1 of a
21: while outlinecolor of cutout right-of cutout touching football is red
22:   --do inner loop; swap or fumble
23:   if data of cutout touching football >
24:     data of cutout right-of cutout touching football
25:   --swap players
26:   assign p1 to gridpos in a of cutout touching football
27:   assign p2 to gridpos in a of cutout right-of cutout touching football
28:   doconcurrent
29:     move cutout touching football to p2
30:     move cutout right-of cutout touching football to p1
31:     move football right cellwidth of a
32:   enddoconcurrent
33:   else --fumble ball to next player in line
34:     move football right cellwidth of a
35:   endif
36: endwhile --inner loop
37: --ball carrier has reached rightful place:
38: set outlinecolor of cutout touching football to red
39: --toss ball back to referee, in preparation for next pass of outer loop
40: move football to top-center of referee
41: endwhile --outer loop
42: set outlinecolor of cutout at position 1 of a to red
```

Figure 53. SALSA Script for Football Bubble Sort Storyboard

7.4.4 Presenting the Storyboard

Having programmed the football bubble sort storyboard, I now proceed to execute and present it. I begin by choosing “Reset at Beginning” on the Present menu, which resets the script to the beginning. Since nothing interesting occurs in the script until after the Storyboard view is populated, I elect to execute the script to the first line past the last “place” command (line 19 in Figure 53) by positioning the cursor on that line and choosing “Run to Cursor” from the Present menu. The first time through the algorithm’s outer loop, I want to walk through the script slowly, so that I can explain the inner logic carefully. Thus, I step forward slowly, pausing to explain the comparison (lines 23–24 in Figure 53) and subsequent exchange (lines 26–32 in Figure 53) of the first two players in line, and using the “Presentation Pointer” tool to point at the storyboard as I explain. Before the two players are swapped, I use the “Markup Pen” tool to circle the two elements that are about to be swapped; Figure 54 shows the ALVIS environment at this point.
Now suppose an audience member wonders whether it might be useful to flash the players as they are being compared. Thanks to ALVIS’s flexible animation interface and dynamic modification abilities, I am able to test out this idea on the spot. As I attempt to edit line 22 of the script, ALVIS recognizes that a change at this point necessitates a re-parsing of the entire while loop. As a result, ALVIS uses a dialog box (Figure 55) to inform me that in order to edit that line, I first have to back up the script to just before the while loop ALVIS. I click on “Yes,” and ALVIS automatically backs up the script to the point just before the while loop (line 18 in Figure 53). I then add four lines of code beginning on line 22:

```plaintext
doconcurrent --flash players to be compared
    flash cutout touching football for 1 sec
    flash cutout right-of cutout touching football for 1 sec
endoconcurrent
```

Figure 54. The ALVIS Environment Before the Players Are Swapped

Figure 55. “Attempt to Edit Previously Executed Line” Dialog Box
I can now proceed with my presentation without having to recompile the script, and without having to start the script over from the beginning.

### 7.5 Related Work

SALSA and ALVIS have key features in common with a family of systems geared toward rapidly prototyping graphical user interfaces. QUICK (Douglas, Doerry, & Novick, 1992) supports a direct manipulation interface for programming graphical applications (which may include animation) in terms of an underlying interpreted language with explicit support for spatial relations. SILK (Landay & Myers, 1995) and PatchWork (van de Kant, Wilson, Bekker, Johnson & Johnson, 1998) are sketch-based systems that support the rapid construction of low-fidelity interface prototypes; the latter’s notion of “patches” is similar to SALSA’s concept of “cutouts.”

Beginning with Brown’s BALSA system (Brown, 1988), a legacy of interactive AV systems have been developed to help teach and learn algorithms (see, e.g., Helttula, Hyrskykari, & Raiha, 1989; Naps, 1990; Roman, Cox, Wilcox, & Plun, 1992; Stasko, 1989; Stasko, 1997). SALSA and ALVIS differ from these systems in three fundamental ways. First, whereas the design of past systems has been based mainly on designer intuitions (so-called “armchair design”), I have made an earnest effort to get out of the design armchair—that is, to root the design of SALSA and ALVIS firmly in empirical data. Specifically, as part of a user-centered design process, I have used observations from several empirical studies as a basis for formulating the functional and usability requirements of the language and system. Moreover, the observations also clearly inspired ALVIS’s conceptual model, which is rooted in “paper-and-pencil” storyboard construction. Finally, before I began implementation, I verified and refined the syntax and semantics of SALSA through a pilot study involving students enrolled in CIS 315.

Second, the AV construction technique pioneered by ALVIS and SALSA differs from those supported by existing systems. To place ALVIS and SALSA into perspective, Figure 56 presents a taxonomy of AV construction techniques. These may be divided into three main categories—algorithmic, spatial, and manual—as described below.

![Figure 56. A Taxonomy of AV Construction Techniques](image)

**Algorithmic.** Algorithmic AV construction involves the specification of mappings between an underlying (implemented) algorithm, and a visualization. In Chapter 2, I called this technique *direct generation*, since it aims to generate a visualization directly as a byproduct of algorithm execution. Variations on algorithmic AV construction include (a) *predefined* techniques, in which one chooses when and what to view, but has no choice with respect to the form of the visualization (see, e.g.,

---

58A noteworthy exception to this is Mukerjea and Stasko’s (1994) work on the Lens system. To assist them in designing Lens’s animation language, they studied 42 actual visualizations developed by course instructors.
Myers, 1983; Naps, 1990); (b) **annotative** techniques, in which one annotates an algorithm with event markers that give rise to updates in the visualization (see, e.g., Duisburg, 1987a,b; Brown, 1988; Stasko, 1989; Stasko, 1997); (c) **declarative** techniques, in which one specifies a set of rules for mapping an executing program to a visual representation (Roman, Cox, Wilcox, & Plun, 1992); and (d) **manipulative** techniques, in which uses some form of direct manipulation and dialog box fill-in to map an algorithm to a visual representation (see, e.g., Mukherjea & Stasko, 1994; Stasko, 1991).

**Spatial.** In stark contrast to algorithmic AV construction, spatial AV construction completely abandons the notion of an underlying “driver” algorithm that generates a visualization as a byproduct of its execution. Instead, spatial techniques aim to specify a visualization in terms of the **spatiality** of the visualization itself. By providing a language in which one can write a procedural specification of a visualization in terms of the visualization’s spatiality, SALSA pioneers a **procedural** approach to spatial AV construction. As a graphical front-end to SALSA, ALVIS supports a **manipulative** approach that allows many components of SALSA programs to be programmed by direct-manipulation. Likewise, two lines of related research explore **manipulative** approaches to spatial AV construction; however, the underlying programming paradigm differs in both cases. First, Michail’s Opsis (1996) supports the construction of binary tree algorithms through the manipulation of abstract visual representations of binary trees. In particular, Opsis users construct a binary tree program (i.e., a visualization) by specifying transformations between abstract program (i.e., visualization) states; this might be called a **state-based** approach. Second, Erwig (1991) and Brown and Vander Zanden (1998) describe systems that support a **rule-based** approach; users construct visualizations by specifying **rewrite rules**, which stipulate how a particular frame of the visualization should be transformed into a new frame.

**Manual.** In **manual** AV construction, one abandons a formal underlying execution model altogether. As a result, and in contrast to the **algorithmic** and **spatial** techniques, there exists no chance of constructing AVs the work for general input; AV execution is entirely under human control. **Storyboarding**, the manual technique on which ALVIS and SALSA are based, involves the use of simple art supplies to construct a visualization. In contrast, Citrin and Gurka (1996) describe a manual technique in which a drawing editor, coupled with off-the-shelf morphing software, is used to produce animations for classroom demonstration. Finally, **drawing and animation editors** such as CorelDraw and Macromedia Director might be used to construct visualizations. For example, in my ethnographic studies (see Chapter 4), I observed several students who used drawing editors to construct their storyboards. In a similar vein, Brown and Vander Zanden (1998) describe a more specialized drawing editor, whose built-in semantics support the construction of data structure drawings that can be interactively manipulated to demonstrate common operations.

The third key difference between SALSA and ALVIS and existing AV technology lies in their support for AV presentation. Almost without exception, existing AV technology has adopted the standard animation control interface pioneered by Brown (1988). As discussed in Chapter 2, that interface allows one to start and pause the animation, step through the animation, and adjust animation speed. ALVIS’s animation control interface goes beyond this interface by supporting both forward and backward execution; this allows the user to back up and review segments of the animation in response to audience requests. In addition, with the notable exception of Brown and Vander Zanden’s (1998) Whiteboard Environment, no existing AV system allows one to annotate an animation as it is running. Finally, no existing AV system supports the dynamic modification of animation; rather, modifying an animation most often entails changing and recompiling source code, which is seldom feasible within the scope of an interactive presentation.

### 7.6 Summary

This chapter has presented SALSA and ALVIS, a prototype language and system that manifest the design implications of the set of hypotheses proposed in Chapter 5. As I have illustrated, taking these hypotheses seriously leads to a design that differs, in three fundamental ways, from that of
extant AV technology. First, rather than supporting highly polished AVs that resemble textbook figures, the prototype supports rough sketches, which require far less overhead to create, and, according to my ethnographic fieldwork, stimulate more relevant conversations about algorithms. Second, the prototype pioneers a novel AV specification technique in which users construct AVs by making use of and explicitly testing the spatial relations among constituent objects. This technique more closely models the way in which humans have been observed to construct and execute storyboards (non-problematically) using simple art supplies. Finally, in departure from existing AV technology’s exclusive focus on supporting AV creation, the prototype presented here focuses extensively on supporting the process of AV presentation. Taking AV presentation seriously leads to the three features not supported by existing technology: forward and backward execution, dynamic mark-up, and dynamic modification.