Appeared in <u>Interactive Learning Environments</u> 4, 1 (1-44)

Software-Realized Scaffolding to Facilitate Programming for Science Learning

Mark Guzdial
Georgia Institute of Technology
College of Computing
Atlanta, GA 30332-0280
(404) 853-9387
guzdial@cc.gatech.edu

Educators' reasons for asking students to program have changed since the early days of Logo and Basic [Solomon, 1986 #262]. Claims that programming alone might improve problem-solving skills or other general metacognitive skills have gone mostly unsupported [Palumbo, 1990 #182]. Today, education researchers are more interested in programming as a medium, as a way of thinking about and exploring disciplines other than computer science [diSessa, 1986 #21; diSessa, 1991 #24; Soloway, 1993 #561]. We are still interested in having students learn *about* programming, because we view programming as an important skill and as a medium of communication. But now we are even more interested in having students learn *through* programming because we recognize that programming is a good lever for understanding many domains.

Harel & Papert [Harel, 1990 #108] are referring to this notion of programming-as-leverage when they write that programming is *reflexive* with other domains, meaning that learning the combination of programming and another domain can be easier than learning each separately. A synergy is created when concepts in another domain are easily reflected in the programming medium. In this case, learning the programming means learning to construct representations for the concepts, which in turn supports learning the concepts and provides motivation for learning to program.

What hasn't changed is that programming is still a hard activity and a difficult skill to learn. Few students will understand programming well enough after completing their first programming courses to be able to write simple programs, let alone use programming as leverage for understanding other domains [Soloway, 1982 #263; Pea, 1986 #146]. The most critical reasons that students have difficult with programming seem to be the following:

- *Assembling programs is hard*. Programming languages have only a few components which are combined in many different ways, and learning to understand the semantic results of different combinations is complex [Schneiderman, 1977 #235]. Understanding how to combine programs to achieve particular goals is a challenge [Spohrer, 1985 #271; Spohrer, 1989 #273].
- *Syntax is complex*. When students try to combine elements, syntax gets confused, which leads to students battling syntax problems as they struggle to understand semantic ones [Perkins, 1986 #196; Johnson, 1985 #311]. When the syntax problems are alleviated, students can focus on the semantic ones [Hohmann, 1992 #260; Soloway, 1993 #267; Anderson, 1989 #479; Garlan, 1984 #583].
- *Students lack an understanding of computational process*. Many students do not understand how interpretation of traditional computer languages works, e.g., where does control flow and how do variables get updated [DuBoulay, 1989 #67]. If students are presented with a simplified or clearer description of the process, they can understand their programs more easily and perform more successfully [diSessa, 1985 #306; diSessa, 1991 #406].

The challenge in using programming to learn other domains is to reduce the aspects of programming which are simply about programming and emphasize the aspects of programming which are reflexive with another domain. For example, a student using programming to explore genetics should not have to be concerned with declaring variables *floats* or *integers*, but on the other hand, being able to define recursive structures is a relevant part of simulating genetic structures computationally. The point is for programming to be an activity which builds upon and adds value to learning in another domain, not to be an activity which simply requires more skills and knowledge which is disconnected from the students' purpose [Oren, 1990 #531; Norman, 1993 #391].

Programming is an activity which leads to a skill, and teachers have techniques for supporting students engaging in an activity and learning a skill. These techniques are called *scaffolding* [Palincsar, 1986 #333; Rogoff, 1990 #220; Collins, 1989 #46; Merrill, 1993 #439]. The goal of scaffolding is (1) to enable students to achieve a process or goal which would not be possible without the support and (2) to facilitate learning to achieve without the support. A critical component of scaffolding is that it be capable of *fading*: a gradual or leveled [Collins, 1989 #46] reduction in the support provided to match individual student ability, particularly as that ability increases with students' learning [Rogoff, 1990 #220]. As students learn the skill, they need less support, so the scaffolding fades.

Scaffolding can be provided through software, which results in *software-realized scaffolding*. By providing scaffolding for programming embedded within the programming environment, we can (1) provide the necessary support for doing and learning the activity and (2) adapt the programming task so that it emphasizes the reflexive nature. *Emile* is a programming environment that provides adaptable software-realized scaffolding to support students as they use programming to learn science, specifically, the branch of physics addressing motion without regard to forces: *kinematics*. Emile's scaffolding is adaptable in that students can fade their scaffolding and choose an appropriate level of support. Students used Emile to program *models* (executable theories of phenomena) and *simulations* (an executing model, being explored through manipulation of parameters) in kinematics. Emile supports students without previous physics or programming background to

- Successfully create models of kinematics and execute these models as simulations, and
- Learn about physics in the process.

Physics in general and kinematics in particular is a difficult topic for students to learn. Students often have misconceptions of the physics of motion [Arons, 1990 #9; Eylon, 1988 #76], and education research has found that these misconceptions are very difficult to correct (e.g., [Champagne, 1985 #329; Trowbridge, 1981 #287; Trowbridge, 1980 #288; diSessa, 1982 #65]). Some evidence suggests that having students program models and simulations is a good way to learn physics. Students can learn through use of pre-existing (e.g., pre-modeled) simulations (e.g., [White, 1984 #283][White, 1984, diSessa, 1982]), but only if students treat the simulations as more than a video game—if they recognize the simulation to be a model of the real world [Richards, 1992 #218]. Having students actually build the model requires students to relate the real phenomena and the simulation, that is, understanding the model as an explanation for the phenomena [Halloun, 1987 #314; Hestenes, 1987 #170; Hestenes, 1992 #171]. There are other ways to get students to relate a simulation to real phenomena, such as through comparison of simulation data to real world data (perhaps gathered with microcomputer-based laboratory probes [Brasell, 1987 #381]). However, as Tinker points out (who is a pioneer in the use of such probes for science education), modeling and simulation (which he refers to together as *model-building* [Tinker, 1990 #284]) requires theory-building, which is an important learning activity and which can easily be overlooked when the focus is on comparison and not explanation. The challenge is in providing a model-building environment that students can use without incurring the difficulties of learning to program.

Emile provides an example of such an environment. Emile was evaluated in a three week summer workshop with high school students. The results suggest that Emile was a success (1) in terms of facilitating construction of interesting physics simulations and (2) in terms of facilitating learning about physics and programming. In this paper, I present:

- Definitions of scaffolding and software-realized scaffolding;
- A description of Emile as an instance of a computer-based learning environment designed to provide software-realized scaffolding;
- The setting for the evaluation of Emile, data collected, and analysis methods used; and
- The results, discussion, and my conclusions.

1. Scaffolding and Software-Realized Scaffolding

The challenge of supporting students engaged in programming for modeling and simulation has two components:

- To facilitate student's programming activity;
- To facilitate student's learning about and through the activity of programming.

The process of teaching an activity and facilitating learning about the activity has been refined for ages in the practice that educators refer to as *scaffolding*. Scaffolding is:

- Support which enables a student to achieve a goal or action that would not be possible without that support;

- Support which facilitates the student learning to achieve the goal or action without the support in the future.

In apprenticeships, for example, the master craftsman provided scaffolding (1) to enable the apprentice to perform the tasks assigned to him and (2) to facilitate the apprentice's learning how to perform the tasks when the master was not available. Similar scaffolding activities occur in apprenticeship relationships today [Lave, 1993 #358].

Collins and Brown have suggested that modern education should model itself on apprenticeship by providing *cognitive apprenticeship* where the skills being scaffolded are more cognitive than those taught in a traditional trade apprenticeship, e.g., Collins and Brown suggest teaching reading, writing, and mathematics using an approach based in apprenticeship [Collins, 1989 #46; Collins, 1990 #42]. They describe in their work the essential qualities of an apprenticeship, such as the elicitation of articulation to encourage reflection and a collaborative, supportive social climate.

Synthesizing the descriptions of scaffolding by Collins and Brown and by other researchers (e.g., [Wood, 1975 #331; Fischer, 1978 #321; Rogoff, 1990 #220; Palincsar, 1986 #333]), I can identify three critical types of support which are combined to provide scaffolding:

- **Communicating process:** A master communicates a process to the apprentice, which typically means demonstrating the process with verbal annotation to highlight key points. Students typically do not know where to begin with a complex process [Jeffries, 1981 #129]. A good master *structures* (often involving *simplifying*) the process to make it easier to communicate. The *presentation* itself may take on many forms (including a simple lecture), all of which are contextualized (or situated) in that the master is providing necessary knowledge for the apprentice who is about to undertake the very same process.
- **Coaching:** When the apprentice is attempting the action or goal, the master watches and makes comments, provides hints, reminds the apprentice of the process which was communicated, etc. Again, since students will typically know little about the process, they cannot be expected to remember the process on first presentation (especially if it is complex). A good coach balances the number and kind of comments between providing opportunities for the student to learn through failure with keeping a student motivated and preventing inefficient exploration of blind alleys [Rogoff, 1990 #220].
- **Eliciting articulation:** The master occasionally asks the apprentice to articulate key concepts about the apprentice's action or goal, e.g., "Why are you doing that?" "Stop! Is that what I told you to do?" "What do you call that?" and so on. The point of eliciting the apprentice's articulation is to encourage reflection – an important cognitive activity which is critical for effective learning [Scardamalia, 1984 #231; Collins, 1988 #45; Schon, 1982 #241].

A critical piece to the concept of scaffolding is *fading*. If the scaffolding is successful, students will learn to achieve the action or goal without the scaffolding. For students to practice the action or goal without the scaffolding, the scaffolding must fade. However, scaffolding should not be all-or-nothing. Instead, scaffolding should be adapted to individual student needs, typically through gradual reductions in scaffolding [Rogoff, 1990 #220]. Students who are more capable (e.g., have more background knowledge, learn the action or goal faster) should have less scaffolding, that is, more fading of the provided scaffolding. The best scaffolding is maximally flexible—providing a continuous range of support. However, discrete levels of support can provide the necessary flexibility such that each student is facilitated in performance and learning without being stifled by too much scaffolding or being left to flounder by too little scaffolding [Collins, 1989 #46].

Scaffolding differs from other forms of educational support in the emphasis on activity and learning through activity. For example, an encyclopedia or other source of authoritative information is typically not scaffolding, while it may be supportive. While information sources may be part of good performance, their role is not to enable that performance or facilitate learning through that performance. Similarly, a supportive social climate is indirect scaffolding. The presence of peers (or communications channels, in a technologically supported community) is not directly supportive. But these peers can provide scaffolding: for example, other people provide multiple models of the process, opportunities to receive coaching, and ears to listen to articulations. If there are many members of the community, there will probably be levels of expertise which can implement *de facto* levels of scaffolding for fading.

Even after distinguishing scaffolding from other forms of educational support, the kinds of support that are scaffolding are still broad-ranging. In actual implementation, I find it useful to distinguish two levels for scaffolding:

- The ***macro*** level is concerned with the *stages* or collections of activities which the student undertakes. The macro stages correspond to those used to describe general problem-solving activity—e.g., Polya's "devising" and "carrying out a plan" stages of problem-solving [Polya, 1957 #201] or the generate and test stages of various cognitive models of how programmers work [Adelson, 1984 #3; Spohrer, 1989 #273]. A good macro level process structure aids students in making strategic decisions about how best to focus on a project over time and complete it [Blumenfeld, 1991 #18].
- The ***micro*** level identifies the individual activities which the students undertake. Polya's problem-solving process is a macro-level-only description since it remains the same for programming or developing a proof of a theorem. Since programming and theorem proving have different individual activities, the processes for those activities differ at the micro level. A good micro level process structure aids students in making tactical decisions that lead them through complex tasks without losing motivation [Blumenfeld, 1991 #18].

Table 1 describes how each of the three kinds of scaffolding differ when the focus is on the macro, stage-oriented level and the micro, activity-oriented level.

The challenge for educational technology researchers is to provide the same scaffolding that a good teacher provides in classroom environments centered on activity. When student learning is to occur in a software environment centered on learning, the environment should provide *software-realized scaffolding*, where the designer of the software is defining and providing scaffolding as the teacher but through the formal mechanism of the software. The goals in software-realized scaffolding are the same as with traditional scaffolding: to facilitate student performance and to facilitate student learning. Resolving this challenge will require defining scaffolding in terms of facilities that can be provided in software.

Many of the successful innovations in educational software interfaces can be viewed as examples of software-realized scaffolding:

- *Communicating Process with explicit checklists or menus.* Explicitly defining the stages and operations of a process with a checklist or a menu can communicate a tremendous amount of information about the process (especially a novice with little knowledge of the process) – it constrains the process to discrete components, it can imply an ordering, and it provides a language for talking and thinking about the process. Examples of communicating process with a checklist or menu include Inquire (a scientific inquiry planning tool) [Brunner, 1990 #362], Framer (a design environment for program frameworks in window-based user interfaces) [Fischer, 1991 #559], and the GPCeditor (a Pascal program design environment) [Soloway, 1993 #267; Guzdial, 1991 #98; Hohmann, 1992 #260].
- *Coaching with critics*. Intelligent agents that watch a user's activity and provide useful comments can be effective in highlighting for a user important perspectives or problems that might have been overlooked (especially a novice who has not yet developed standards for applying to her own work). Critics have been used to highlight design principles which have been broken in the user's artifact (e.g., in Janus, a kitchen design environment [Fischer,

1990 #319]) and to highlight when a user has strayed from a pre-defined successful path, e.g., in GIL (a graphical tutor for Lisp) [Merrill, 1992 #440; Merrill, 1993 #439] and in the ACT* tutors [Anderson, 1989 #479; Anderson, 1990 #515].
- *Eliciting articulation with pre-defined prompts*. At various points in the process at which reflection is most profitable, students can be presented with a prompt asking them to articulate what is most profitable to reflect upon. The prompts are based on the questions that a good teacher might ask at a similar point, e.g., "What is your goal for the next piece of code that you're going to write/construct?" before the student identifies a piece of Pascal program to add to her program (GPCeditor) and "What are you going to when you continue this tomorrow?" before the student ends the program (Inquire), which is also shown again to the student at the beginning of the next session.

A complete implementation of software-realized scaffolding should also include fading of the scaffolding. Using the distinctions made in the HCI community (e.g., [Suikaviriya, 1993 #554; Schneider-Hufschmidt, 1993 #405]), we can define *adaptable scaffolding* (i.e., scaffolding which can be changed or faded by the user) and *adaptive scaffolding* (i.e., scaffolding which changes or fades based on an internal decision process). Anderson has explored intelligent tutoring systems that offer a choice of immediate or when-requested coaching [Anderson, 1990 #515], which is adaptable scaffolding, but with an all-or-nothing approach. Riel et al. [Riel, 1987 #219] propose a form of adaptive scaffolding which they call *dynamic support* to detect and automatically adapt interactions to the student's ability. While adaptive scaffolding (or intelligent fading) is clearly desirable, current interface technology is not yet up to the challenge of changing the scaffolding in such significant ways for a user who is rapidly changing in ability (as does a novice when starting a new domain or new environment). What success there has been in determining novice user's ability is offset by the amount of time needed to make the determination—much longer than can be afforded in order to provide adequate scaffolding [Vaubel, 1990 #494].

2. Emile: Software-Realized Scaffolding for Modeling and Simulation Activities

Emile facilitates student programming for modeling and simulation by implementing software-realized scaffolding in the programming environment. The challenge in creating Emile was to provide (1) the full set of scaffolding types (i.e., communicating process, coaching, and eliciting articulation) with (2) adaptability. Thus, Emile serves as an example of a complete implementation of software-realized scaffolding. Emile's scaffolding is adaptable by a student or teacher, as will be seen, with the explicit suggestion that the fading of scaffolding be accompanied by discussions between students and teacher of when scaffolding might be appropriately changed.

This section presents
- An overview of Emile.
- A description of Emile as a learning environment utilizing the range of software-realized scaffolding: Communicating process, coaching, and eliciting articulation, with fading.

2.1 Overview of Emile

Emile is an environment where students build models and test them as simulations. They are prompted to write about what they are doing in a *Design Notebook*. The Design Notebook is also where students define the components of their model. The components are assembled in a *Project Window* which is where the simulation ran. In addition, a number of supporting facilities are provided, which can be modified from a *Preferences* page in the Design Notebook.

Figure 1 is the view that a student has after choosing to create a new, empty project from the File menu.

- The main window (largest, seen at front) is the Design Notebook which contains all of a student's programming representations, all of a student's articulations about the project, and descriptions of all the components of the project – each on a separate page of the Notebook. The Notebook organizes the various pieces of a project.
- The window which can be seen behind the Design Notebook and to the right is the Project Window where the components of a student's project are assembled and tested. When a student completes her project, the Project Window (stored in a file separate from the Design Notebook) can be given away to others to execute without Emile – it is a standalone component[1]. Figure 2 shows a Project Window that is a simulation of two objects which fall as if under the influence of gravitational fields of different intensities.
- The window to the far right of Figure 1 is the Recent List, one of several Notebook *navigation tools* available in Emile. The Recent List shows the name of each page visited in an Emile session (seeded with some important pages at the start of a session). A mouseclick on any name turns to that page. The right and left arrow keys at the top left of each page allow page-to-page navigation in the Notebook. The first page of the Notebook (seen in Figure 1) is the Table of Contents which is both a textual representation of all the pages in the Notebook and a navigation tool: Clicking on any page name (one per line in the Table of Contents) turns to that page.
- The five main *menus* in Emile (seen at the top of Figure 1 are named Initial Review, Decomposition, Composition, Debugging, and Final Review for five programming activity stages that define the process that is communicated to students in Emile. All the operations associated with the activities in each stage appear under the menu named for that stage. For example, the operation to create a new planning articulation page appears in the Initial Review menu, to correspond to the planning activity associated with the initial review stage.

Figure 2 is a screenshot of a Project Window depicting a sample of the kind of program that a student might create with Emile.

- The graphical objects Positive Gravity, Weaker Gravity, Compare Buttons, and Clear Graphics are all *buttons* which can be clicked on to generate program *behavior*. For example, clicking on Clear Graphics clears the screen of the small circles which are used to trace the trail of the buttons Positive Gravity and Weaker Gravity. Each button has a page to itself in the Design Notebook defining its appearance and behavior.
- Positive Gravity and Weaker Gravity are the focus of the *model-building* in this program. Each of these graphical objects can be clicked down upon (using the mouse cursor), dragged to the top of the screen, and released – after which the object falls as if it was attracted by gravity.
- The *fields* in upper right corner hold text and numeric data: Labels for the fields, and the values for the objects' vertical position on the screen, velocity, and simulation time. Each field in the Project Window also has a page in the Design Notebook which describes its appearance and behavior.

2.2 Software-Realized Scaffolding in Emile

Emile is unique in the depth and breadth of scaffolding it provides to the programming student.

- The full range of scaffolding (communicating process, coaching, and eliciting articulation) are all provided. There are other programming environments which perform some of these roles (e.g., Boxer structures and presents a model of computation [diSessa, 1985 #306;

---

[1]The Project Window does require Apple HyperCard or HyperCard Player to execute, but these are readily available. The task of creating simulation programs which could be used apart from Emile seemed to enhance the authenticity of the task

diSessa, 1991 #406] and the LISP Tutor coaches students through a process [Anderson, 1989 #479]), but few provide all three.
- Fading of scaffolding is supported. Fading is difficult to provide since it means creating multiple interface paths to multiple functionality. While some programming environments have support that can be turned on or off (e.g., the LISP Tutor could have advice on demand rather than immediate), few implement multiple levels of adaptation.

The goal of the scaffolding is to facilitate learning and performance. In particular, Emile addresses the three critical problems that students have with programming:
- Assembling programs is hard,
- Syntax is complex, and
- Students lack understanding of computational process.

The subsections below present (1) each type of software-realized scaffolding in Emile and fading of that scaffolding. Table 2 summarizes the software-realized scaffolding in Emile.

2.2.1 Communicating Process

Communicating process, as described previously, has two parts to it:
- *Structuring* (often *simplifying*) the process that the student will be performing, and
- *Presenting* that process to the student.

STRUCTURING: The process that is being communicated with Emile's scaffolding is *programming for model-building:* Creating models in a computer-understandable language of physical phenomena and testing the models through simulations. Specifically, Emile supports the creation of models with graphical representations, which has become the *de facto* standard in modeling and simulation [Pidd, 1989 #340; Tinker, 1990 #284; Earnshaw, 1992 #410]. In creating Emile, I defined both specific macro and micro definitions of this process which structured the students' activity, then developed presentation mechanisms to communicate the defined process.

The macro process supported in Emile is based on the models of Polya, Adelson, Spohrer, Hestenes, and others. Most directly, the macro process is based on a process supported successfully by another scaffolded programming environment, GPCeditor [Guzdial, 1992 #97; Soloway, 1993 #267; Hohmann, 1992 #260]. Emile extends the macro process supported in GPCeditor with explicit stages for reflection. The stages in Emile's supported macro process are:
- **Initial Review**: Students in this stage undertake activities to help understand the problem and plan their future activity. Initial Review is expected to take place both at the beginning of a project and at the beginning of a model-building session. Initial Review is analogous to Polya's "understanding the problem" stage.
- **Decomposition:** Students in this stage determine their program goals and choose or create components to achieve those goals. Decomposition is analogous to Spohrer's Generate phase.
- **Composition:** Students in this stage assemble components into executable models. Composition is defined as an explicit stage of program generation in both GPCeditor and Emile because of the difficulties students have in composition activities [Spohrer, 1985 #271].
- **Debugging:** Students in this stage test their models as simulations. Debugging is an important stage in many different models of problem-solving, model-building, and programming because of its role in providing feedback to the student. If debugging leads to an impasse that forces *successful* reconsideration of the model (and continued work at the Decomposition or Composition stage), we expect students to learn [Spohrer, 1989 #273].

- **Final Review:** Students in this stage review their model-building activity—saving pieces for later reuse or noting reflections in a journal. Final Review is analogous to Polya's "looking back" stage.

Within each of these macro process stages are micro process activities. Some of the micro activities are articulation activities which are described in a later subsection. Most of the micro activities are programming activities. The metaphors and programming structures presented in Emile simplify the activity of programming to make it easier to define for the students the process they are to follow in model-building with Emile.

There are many programming structures for manipulating graphical elements—none of which are particularly easy to program (see [Lee, 1993 #560] for a review of graphical user interface programming structures and environments). Apple's HyperCard has one of the simplest structures for programming with graphical objects [Nielsen, 1991 #313; Goodman, 1977 #116], though it is still not simple enough that students can readily program something as complicated as a model with it [Decker, 1990 #117; Lehrer, 1992 #328; Ambron, 1990 #126]. The structure for programming with graphical objects that has been used in Emile is a simplified form of HyperCard's structure.

Emile's programming structure has five components, which extend from the high-level *groups* and *goals* which organize components into related modules; through *buttons* and *fields* which provide the graphical representations in the model; finally to *actions* which define the low-level details of the simulation.

- *Groups* and *goals* collect and relate other components. Thus, groups and goals serve as the modularization structures in Emile, and modularization can be an important tool in simplifying programming [Parnas, 1972 #191]. Students create a goal to define an objective: a statement of intent. A group can encapsulate lower-level components: buttons, fields, actions, and also goals. Components are *linked* to a group to identify relations. A group can be associated with a goal by *matching* the group and goal.. Because a group can also contain a goal (which in turn can be matched to another group), a hierarchy of goals and groups and subgoals and subgroups can be created to reflect the program structure. Activities on groups and goals are:
  – Composing and uncomposing an entire group of components.
  – Linking and unlinking a component from a group.
  – Declaring a match or removing a match between a goal and a group.
  – Creating new groups or goals.
- *Buttons* and *fields* are the graphical objects in Emile, which normally appear in the Project Window but are defined in the Design Notebook. Buttons are graphical entities which can be programmed to respond to mouse clicks. Fields are graphical entities into which text can be entered or displayed. When a button is programmed to execute some action, it is said to have a *behavior* which is created from one or more actions. A behavior is executed upon receipt of a user-initiated *trigger* event – by default, a mouseclick (a *mouseup* event, in the terminology of HyperCard which Emile inherited). Activities on buttons and fields are:
  – Tailoring the graphical appearance of the button or field (e.g., changing the shape of a field, specifying an icon for a button, changing the position or size of a field or button.)
  – Compose and uncompose a button or field from the Project Window.
  – Creating new and copying buttons or fields.
- *Actions* are the lowest level of abstraction[2]. An action is one or more programming statements (in the programming language HyperTalk) collected to achieve a particular purpose. Figure 3 shows an action, *Accelerated Motion*, which implements the

[2]One might imagine a level of abstraction between buttons and actions where the domain-oriented name of an action and its slots are visible, but the HyperTalk code itself is invisible. This would be a welcome addition which could further insulate the model-builder from unnecessary programming details.

computation of velocity and position for an object freely-falling. An action can have certain expressions identified as tailorable through a mechanism known as *slots*. A slot is a named expression position which is filled by the student at the time of use. For example, in figure 3, a slot named *Number for acceleration* will be filled with an expression (e.g., a constant, a reference to a field) which will specify the acceleration due to gravity for the model. Slots are not unlike terminal holders in structure editors (e.g., [Garlan, 1984 #583]), but in Emile, only those holders which are semantically significant for the action are fillable slots (e.g., if an indexed loop in an action always starts at 1, no slot will be defined for starting value on that loop—the action will simply contain the constant 1.) Activities which can be performed on actions are:

– Composing and uncomposing an action from its use in a given button's behavior.
– Positioning an action within a program[3] to create an order which achieves a particular goal. Positioning actions will require users to read and understand the semantics of program components.
– Emptying a slot of its current value and filling a slot with a new value. Filling slots will require users to read and understand slots in the context of their actions in order to achieve the users' objectives. (Slots that are filled may also be queried to retrieve their original slot name without emptying the slot.)
– When scaffolding fades, students can create new actions and new slots, and also compose actions into fields, but not at first.

To describe how Emile simplifies programming activity, I compare Emile to HyperCard (which is already a simplification of more traditional programming). The structure in Emile is based on the programming structure provided in HyperCard. HyperCard provides more flexibility, but does so through additional levels of complexity and without modularity mechanisms:

- HyperCard lacks a structure like groups and goals for relating similar components and manipulating them as a set. Instead, it provides a visual encapsulation system—all buttons and fields visible at once on the screen are assumed to be related and can be manipulated as a set by copying or pasting the *card* or *background* which encapsulates the elements. HyperCard's structure does not allow for hierarchical decomposition to define lower level relations and reduce complexity [Parnas, 1972 #191].
- HyperCard uses its encapsulation mechanism to also define an inheritance mechanism where functionality for low-level components (buttons and fields) can be defined in encapsulating components (cards or backgrounds). While inheritance reduces complexity for building large systems, Emile does not support inheritance because of the added complexity it adds to understanding models [Kay, 1993 #135].
- Each button and field in HyperTalk can respond to a large number of user events or *messages* (e.g., mouse movement over a button versus a press of mouse button versus a release of a mouse button) and any number of programmer-defined messages. Emile only allows a limited number of events to be handled by a button or field and no programmer-defined messages. In fact, in the default scaffolding, only buttons could respond to messages and only the message corresponding to a mouse button click.
- Programs in HyperCard are written in HyperTalk, a procedural programming language with a phrase-oriented (e.g., wordy) grammar [Goodman, 1977 #116; Nielsen, 1991 #313]. While such a language is terrific for providing an easily readable format, the large amount to be entered for basic functionality requires a lot of syntax to be learned and provides ample opportunity for syntactic errors. Emile does not require students to enter HyperTalk (though it does permit it)—instead, students can assemble and tailor existing actions (which are written in HyperTalk).

[3]In the version of Emile described here and used in this study, the positioning operations are cut and paste of actions. In future versions, actions will be positioned by dragging and dropping.

Thus, the process structure that Emile communicates to the student is much simpler than in traditional programming. Specifically, the process structure helps in addressing the three critical problems of student programming identified earlier: (The coaching and eliciting articulation scaffolding also help to support this structure and address these problems.)

- Assembling programs is hard, but placing graphical objects on the screen, composing and ordering actions, and filling slots are much more straightforward. Certainly, some programming problems are still present (e.g., how do I get a button to move horizontally while falling?), but those are the problems that we want to students to face—these problems are model-building issues that are reflexive with programming issues.
- Syntax is complex, but Emile make syntax relevant only for reading because the syntax is embedded in actions and actions can only be manipulated in specific ways. Students must understand the syntax in order to use them effectively, but they don't need to be able to generate those actions. The wordiness of HyperTalk then becomes an asset rather than a liability, since the extra words seem to aid in readability [Nielsen, 1991 #313].
- A computational process is made explicit through the notions of triggers and behaviors. It is an impoverished process compared to HyperCard's, however the reduced computational process might be considered a benefit to understandability and approachability [Fischer, 1978 #321].

PRESENTATION: Given the structure described above, the next piece of communicating process is to present that structure. Emile uses a variety of interface strategies to present the process structure. The goal of these strategies is to perform the same task of a good teacher: To communicate what is to be done and to provide examples of how to do it.

The macro level process in Emile is presented through the menu system and the Design Notebook *Design Stage* pages. Figure 4 shows an example Design Stage page with the menu bar appearing across the top of the figure.

- The menu bar lists the design stages defined earlier, listed in the order in which one would expect them to be used: Initial Review, Decompose, Compose, Debug, and Final Review. The menu bar serves as a constant presentation of the process structure defined for Emile.
- For each of the design stages, there is a Design Stage page which is available in every Design Notebook. The Design Stage page describes the stage, identifies the kind of actions which collect under that stage, and defines each action.

Micro level process is presented through four components:

- The menu system which clusters the available activities into groups corresponding to macro level design stages.
- The Library for providing example program objects.
- The representations of objects and relationships between them.
- The Design Notebook for organizing components, tools (including representations), and prompts and articulations.

The activities associated with each macro level process stage appear under the menu named for the process stage. As can be seen in Figure 4, the menu under Decompose lists those activities which are associated with the decompose design stage. Figure 5 summarizes all the menu activities in Emile by presenting all of the design stage menus. The bullet list and figures below present examples of how each design stage menu might be used in actual use, where the hypothetical scenario is a student creating a project where some objects are subject to a weaker gravitational field than others [Guzdial, 1993 #104].

- *Initial Review*: The initial review stage is defining a Project Description (either starting with one from a set of examples or creating a new one from scratch) and of making plans. When a students selects the *Make Plans* menu item, a new page in the Design Notebook is created with prompts for daily plans (figure 6).

- *Decomposition*: The decomposition stage is the set of activities to create new program objects, duplicate objects, copy objects from the Library of objects (described below), and link groups, and match goals. Students are expected to define goals (see example in figure 7), define groups, then match the group to a goal (figure 8) and link related components to the group.
- *Composition*: The composition stage is when students assemble the complete project: composing buttons, fields, or complete groups onto the Project Window (figure 2); composing actions into buttons; tailoring the appearance of buttons and fields (figure 10 and 11); and filling and emptying slots (figures 12 and 13).
- *Debugging*: The debugging stage consists of the activities making a prediction, testing the program (figure 14), and analyzing the program by tracing it and slowing it down.
- *Final Review:* The final review stage is where students create a journal entry (figure 15), generate an index for the Design Notebook to aid future readers, and copy particularly useful components into the student's Library.

The Library is a key component of the micro level process. The Library appears as a Design Notebook without the articulation and design tools pages (figure 16). It provides complete, useful groups and program components (buttons, fields, and actions)—over 100 entries. Table 3 lists the components in the basic Emile library. These components range over the domains of physics and multimedia and include both high and low level (i.e., more abstract versus closer to the domain) components, such as:

- The action *Accelerated Motion* which computes the velocity and location of a freely-falling object.
- The action *Play a QT in Window* which plays an Apple QuickTime™ format digital video in a floating window.
- The actions *Test for equality* and *Test for greater-than* which are conditional statements (If-Then-Else) with the corresponding relational tests built-in to the action.
- The buttons *Positive Gravity* and *Droppable Object* which each simulate one-dimensional projectile motion by falling down the screen with acceleration. *Positive Gravity* has its behavior defined with high-level actions such as *Accelerated Motion*, while *Droppable Object*'s behavior is defined with more low-level actions such as *Repeat up a range* (an indexed repeat loop).
- The group *Gravity Simulation* which includes *Positive Gravity*; the fields *Velocity, Position,* and *Time*; and all the associated actions. To construct a one-dimensional projectile motion simulation, students need only copy the *Gravity Simulation* group, compose it, and test it. Since the group actually consists of all the component program objects, students may then tailor and add to the simulation to get the desired functionality.

The library serves three important functions:

- First, it serves as a starting place. By providing already existing objects for use and tailoring by students, Emile reduces the number of micro-level activities required to create a simulation and thus reduces complexity. As is seen in a later section, students did use the library frequently to start their projects.
- Second, it serves as a demonstration or presentation of model components. The Library components are examples (or cases) of working, useful elements. At one point during the evaluation workshop for Emile, a student kept both his Library and his Design Notebook windows open at once and compared them back and forth to see if the programs he was writing were like those in the Library.
- Third, students can (and do, occasionally) save out to the Library particularly useful components for later reuse in a future project.

The programming objects used in Emile are navigated, described, and manipulated using a variety of representations. The representations provide another presentation of the program structure for students. Three representations are particularly important:

- *Table of Contents*: The Design Notebook and Library each begin with a hierarchical list of each of the pages in the Notebook (e.g., Figure 1). All pages in the Notebook are listed: from Plans and Project Descriptions, to Buttons and Actions. Pages are listed one-per-line of the list, with groups (such as "Design Stages" or "Buttons") indicated with indentation. Clicking on any line opens that page in the book. (Only one page can be visible at once.)
- *Project Chart*: For each component in a project, a corresponding icon is created on the graphical Project Chart (figure 16). The icon is named with the name of the component. Icons on the project chart can be organized in any way desired by the student. Decomposition (match and link relationships) or composition (composed-within relationships) can be overlaid as arcs on the icons. Students can use the Project Chart both to have presented the current structure of their project or to use the two-dimensional space to create their own structure.
- *Individual Component Pages:* Each component in a project has a page on which the characteristics of that component are presented and can be modified. For example, Figure 18 shows an example page for specifying a button's characteristics:
  - The name of the button is at the top
  - Below the name is the indication that the button is currently not linked to any group.
  - The Show Me button causes the button to flash on the Project Window to identify its location should the student somehow lose it (e.g., make it invisible, or layer it behind).
  - The student can modify the Appearance of the button by choosing the Appearance view of the page (Figure 11). In Figure 18, the Behavior view is selected. The student can also choose the How it Works view for a description of the button.
  - The middle of the page shows that the mouseUp trigger is currently selected.
  - The mouseUp behavior is seen at the bottom of the button's page. This is the same behavior as in the Positive Gravity, but references to the Positive Gravity button are changed to references to the Weaker Gravity button. Each line is marked with the action to which it belongs and the line number within that action. Underlined expressions are slots.
  - The actions and variables in the behavior are listed above the behavior.

Finally, the Design Notebook is the metaphor for presenting the structure of the entire Emile project. The Design Notebook helps in resolving the problem of how to handle the complexity of so many supporting features. The metaphor of a Notebook is enhanced with a spiral binding graphic on each page, a Table of Contents, and an Index page. All the prompts for articulation, representations[4], design stage presentations, and components in Emile appear as pages in the Design Notebook. Only one page can be visible at any time. Several navigation methods are available for moving between pages:

- Clicking on the name of a page in the Table of Contents turns to that page.
- Each page of the Notebook contains buttons for turning forward or backward a page or for jumping to the Table of Contents.
- The Recent List is a floating window showing the most recently visited pages. Clicking on any name in the list turns to that page.
- Students can jump to the page for the selected component icon from the Project Chart.

2.2.2 Coaching

Coaching is the support provided while the student works through a process, in response to the student's actions. Coaching in Emile serves to remind students of the macro level process for

---

[4]The larger size of the Project Chart is explained as a "fold-out" page.

Emile (i.e., design stages) and the micro level process for Emile (specifically, the use of goals and groups). Two kinds of coaching are implemented in Emile:

- **Stage prompting:** When a student first[5] starts Emile, the five process stage menus are all disabled. To enable a menu, the student must visit the corresponding page in the Design Notebook which describes the stage, the menu, and the operations on that menu and in that stage. This coaching forces students to be reminded of the definition and activities in the macro level stage before activity can be selected from that stage.
- **Top-down design enforcement**: Some research in programming suggests that students will learn to program better using a top-down design methodology, where students must articulate a purpose for a component before defining a component, and then move on to define the subcomponents in a sequence of articulations-then-mechanisms [Jeffries, 1981 #129; Soloway, 1986 #265]. Emile provides enforcement of top-down design by preventing students from manipulating actions until all components have purposes (goals) associated with them (i.e., linked to groups that all components belong to).

There are other kinds of coaching support which have been described in the literature on human-computer interfaces and on educational technology. For example, critics would have been a useful addition to Emile, not only to critique the students' developing designs [Fischer, 1990 #319], but also to coach them through adaptation of the scaffolding (described later in the section on fading). However, critics can be expensive to implement on several measures (e.g., computational processing cost, the requirement for a more elaborated process description, and the requirement for software to analyze user behavior and compare to the process description), and I chose to forego that implementation in Emile.

2.2.3 Eliciting Articulation

Eliciting articulation is encouraging students to articulate and reflect on their project and their learning. Eliciting articulation in Emile has a macro and micro level distinction to it. Macro level eliciting articulation are the prompts which occur at specific points in the design process. Emile provides several of these macro prompts:

- **Beginning a project:** As a student begins a project, she is expected to create a *Project Description* which states a project's purpose.
- **Beginning a session:** As a student begins a new session with Emile, she is expected to create a *Plan page* (Figure 6b) which prompts the student to describe what she is going to do in the given session.
- **Before defining components**: Before defining or copying a component, a student is expected to define a *goal* (introduced earlier) which prompts a student for the purpose of a component and alternative ways of achieving the given purpose.
- **Before testing:** Before testing a complete program, a student is expected to create a *Prediction page* which prompts the student to describe the expected behavior of the simulation or program.
- **After testing:** After testing, a student is returned to the prediction page (if one was created). She is shown a list of user actions (e.g., which buttons were clicked upon) and is prompted to describe what went wrong and how she might correct it if the actual behavior didn't meet the prediction.
- **Ending a session**: Before ending a session, a student is expected to create a *Journal page* where she is prompted to describe the session's events, summarize what happened, and make initial plans for the next session.

[5]Stage prompting is usually one of the first scaffolding that students fade, so it is usually only new students that are using stage prompting.

The micro level eliciting articulation are the prompts which are pervasive throughout the model-building process: Naming components. Papert has emphasized the importance of naming for student programmers as a means to connect their understanding with the computational objects being used to model that understanding [Papert, 1980 #168]. For the same reason, students must name even primarily graphical objects such as buttons and fields, besides goals, groups, and actions. The naming is critical in helping students understand the role of the components, as seen in slot names like *Number for Acceleration* and field names such as *Velocity* and *Time*. The act of naming is important, too, because the names appear so often – in references in behaviors, in representations, in the Project Window, and as Design Notebook page names. Thus, naming is considered to be an important elicitation of articulation which influences low-level activity so is seen as a micro level support.

2.2.4 Fading

One of the goals of Emile is to provide adaptable scaffolding – software-realized scaffolding which fades. There are two roles for fading:
- To support individual students as they change over time, and
- To support different students with their individual strengths and weaknesses.

Not all of Emile's scaffolding fades, but much of it does. Emile provides fading in three ways:
- **Through less use of voluntary supports.** Much of Emile's scaffolding is not enforced. For example, students are not forced to use the library of components, multiple representations of components, groups, or articulation and reflection prompts. This means that students can choose to use fewer of these prompts as they develop skills to supplant the scaffolding.

  For example, the Library is a voluntary support: Students do not have to use the Library. As they develop the skills to create their own components, they need not use the Library. Continually revisable software-realized scaffolding is provided by allowing Library and student-created components to co-exist in the same projects. In this way, a student can choose how much he or she wishes to rely on the library components in projects.

- **Through student-selected levels of supported use.** For other scaffolding, fading occurs through manipulation of the Preferences page in the Design Notebook (Figure 19). Fading for these scaffolding is not as gradual as it is for voluntary-use scaffolding. Instead, different combinations of Preferences settings introduce scaffolding *levels*.

  For example, actions and slots fade in levels. When students begin using Emile, they are limited in their manipulation of text programming to positioning actions and filling slots. Students can choose several levels of manipulation beyond that, some of which are:
  – Students can choose to fill slots with expressions, which requires some knowledge of syntax but still provides extensive support.
  – Students can choose to create actions, but still use composition operations for combining actions and for filling slots.
  – Finally, students can directly edit behaviors.

  Use of behaviors in buttons and fields are also implemented in levels. When students begin using Emile, they can only create behaviors for buttons on a mouse click user event (*trigger*).
  – Students can also choose to add behavior to fields.
  – Students can also choose to add behavior to other triggers (e.g., a behavior to be activated as soon as the mouse passes over the boundaries of the button).

An important benefit to Emile's implementation of levels is that scaffolding can be faded or returned at any point. Students return to greater levels of support by simply changing back the Preferences page. For example, Emile supports students intermixing actions from the Library with directly edited statements (Figure 20). Students can fade the scaffolding in order to directly type some lines of a behavior, then turn the scaffolding for action-oriented editing back on in order to grab some useful actions from the Library. Thus, a student can switch back and forth between actions-oriented behavior creation (for unfamiliar parts of a behavior) and directly editing behaviors (for parts that the student is comfortable with typing directly). Rogoff says that real scaffolding often has this characteristic—that it fades and returns several times during learning [Rogoff, 1990 #220].

- **Through immediate stopping.** Some scaffolding ends immediately without fading. Stage prompting and top-down design enforcement both fade like this: It's either on or off. One could imagine more flexible top-down design enforcement. For example, instead of outright prevention of composition until top-down decomposition had occurred, partially faded top-down design enforcement might suggest other decomposition operations or prompt the student for a justification. However, I did not implement that level of flexibility in Emile.

Table 4 summarizes the kinds of fading implemented in Emile for each category of software-realized scaffolding:

- **Communicating Process:** Macro support for communicating process does not fade in Emile. While one might imagine creating a computer-based environment that explicitly supports different macro-level processes (e.g., a planner versus a bricoleur orientation [Turkle, 1991 #289]), Emile only supports a single macro-level process.

  Micro support is quite adaptable. While the individual activities (as presented in menu items) do not change, actions and slots can be faded in levels (as mentioned above), behaviors in buttons and fields can be faded in levels (again, as described above), various navigation tools in the Design Notebook can be adapted, and functioning in some representations is adaptable (i.e., whether decomposition and composition relations can be constructed graphically in the Project Chart without using menu items). There is no adaptable scaffolding associated with goals and groups or the Library, but since their use is not required, their use can be voluntarily reduced as the student desires.

- **Coaching:** All coaching in Emile is either on or off – no levels are supported. One can imagine providing coaching that is adaptable, e.g., critics that comment less often to allow more exploration by a user, but this was not implemented in Emile.

- **Eliciting Articulation:** Students are only required to enter names for objects—none of the prompts for articulation are required. This allows articulation prompts to fade through voluntary less use. One might imagine a graphical programming environment where objects are optionally named or in which only certain objects need to be named, which would allow for additional flexibility in articulation scaffolding. Reference in such a name-less environment might be established through a drag-and-drop interface and be represented as arcs (some of which is explored in [Petre, 1990 #459; Petre, 1993 #453]). Such options were not provided in Emile in order to mesh easily with a more traditional textual programming language.

A potential weakness in Emile's implementation of scaffolding is that control over the fading of scaffolding rests in the student's hands. While some researchers argue that scaffolding is always at least partially in the student's hands and that scaffolding fading is a negotiated process [Rogoff, 1990 #220], other researchers point out the weaknesses in students' metacognitive skills which reduces their ability to make decisions about scaffolding for themselves [Farnham-Diggory, 1990 #79]. As an exploration of the approach of offering fading for a wide range of software-realized scaffolding, I chose to take the extreme position of allowing students to control their own fading, with only the classroom instructor's review to serve as a check.

3. Evaluation of Emile

Emile was evaluated to address three questions and corresponding hypotheses:

- How do students use software-realized scaffolding? In particular, do they fade the scaffolding and use the scaffolding differently (1) over time and (2) between students? My hypothesis was that scaffolding would be used by students (e.g., voluntary supports would be used) and that this use would change over time.
- Does the scaffolding successfully support performance of the process: programming for model-building? My hypothesis was that students would be able to build models and execute simulations in Emile—in fact, I expected them to complete several programs in a relatively short period of time.
- Does the scaffolding successfully support learning of programming (learning *about*) and physics (learning *through*)? My hypothesis was that students would learn both about programming and physics.

The following subsections describe the workshop in which Emile was evaluated and the students who took part in the evaluation. Three sources of data were collected to address the evaluation questions:

- Log files were collected to note use of scaffolding and adaptation of the scaffolding.
- Student projects and Design Notebooks were collected to assess performance.
- Clinical interviews were taken before and after the workshop to assess learning.

3.1 Description of workshop and students

Emile was evaluated in a three week summer workshop with five volunteer high school students from Ann Arbor, Michigan, and local school districts. The workshop was advertised as part of Ann Arbor's summer school program. The workshop was named *Teaching Physics with HyperCard* and was described to students as an opportunity for students to (1) learn programming skills while (2) improving their physics skills. The focus of the workshop was to create software that high school physics students could use. Students were offered high school science credit for taking the course. The pre-requisites for the course were previous high school physics coursework or permission of the instructor. Students were not required to have previous programming experience. The workshop ran for three weeks in August for three hours a day, five days a week, which is comparable to the length of other similar workshops to explore science education initiatives with high school students [Champagne, 1985 #329; Tinker, 1990 #284].

Five students volunteered to attend the course. In general, the students were fairly representative of high school students in Ann Arbor (a middle-class, Midwestern community) except for gender and motivation:

- All five were male: Four were white, and one was Hispanic.
- Two students were from different Ann Arbor private schools, two others were from different Ann Arbor public schools, and one student was from a town adjacent to Ann Arbor.
- Grade point averages were not available for all students. However, pre-workshop interviews suggest that the students were not remarkable in their understanding of physics.
- Though I took no measure of motivation, students must be considered more motivated than in a traditional class to take a three week, three hour a day workshop during the summer.

The ages, grades, and previous programming and physics experience of the five students are described in Table 5. None of the students had previous high school physics coursework experience, so that requisite was waived. Only two had previous significant programming experience (students S and M), and no students had previous high school physics.

For most of the students, their primary interest seemed to be programming and use of computers, not the physics. For one student, C, neither the programming nor the physics was the draw—he needed high school science credit to graduate, and his school district accepted the summer Emile workshop as sufficient. In general, I think it is fair to say that the students did not enter the workshop strongly motivated to learn physics.

The workshop room was equipped with ten multimedia computer systems. Half were Apple Macintosh II computers, and the other half were Apple Macintosh SE computers with accelerator boards. All ten were equipped with 19-inch black-and-white monitors. Videodisk and CD players and sound digitizers were available for all students. Each computer's hard disk was pre-loaded with Emile, a drawing program (for creating graphics), a collection of example Emile projects, and MediaText (a multimedia composition tool, which students used for previewing videodisks.) The Macintosh II computers also were loaded with sample Apple QuickTime™ digitized video movies.

Students worked on three model-building and one multimedia presentation project. I was the instructor for the course. The flow of the course was cyclical. For each of the projects, there was one iteration of a cycle of instructor presentation, student work, and instructor review.

- On the first day of a new project, I gave a presentation on the physics that the students would be using on the next project and a demonstration of using Emile for creating some of the project elements.
- Students would work on the project for three days (including the rest of the day after the presentation and demonstration.) During their programming and model-building activities, I was available for questions. The only articulation prompt I insisted they use was a daily journal entry. I encouraged use of others, but did not require them.
- On the third day, students would complete a Project Evaluation for their programs, then demonstrate their programs to me. The Project Evaluation explained the purpose of the program: What a student should learn about physics from using this program and what the student would do with the program. Students received no grades for their individual projects[6], but each student did discuss the projects and evaluations with me. One of the goals of this discussion was to address misconceptions reflected in the evaluation. For example, if a student wrote that "gravity moves" on their Project Evaluation, I would explain that gravity did not move, and we would discuss why the student thought it did.

Here are the abbreviated project descriptions as they were given to students.

- **Project 1: Simulation of projectile motion in one dimension.** Starting with Droppable Object or Positive Gravity buttons from the library (buttons which can be dropped and which fall as if under the influence of gravity), create at least two new gravitational objects (buttons) such as an object with negative gravity, an object with stronger or weaker gravity, or an object with gravity to either side of the window.
- **Project 2: Demonstration of projectile motion in two dimensions.** Build a presentation on what projectile motion under the influence of gravity is like in two dimensions. Use at least one text field explaining motion and gravity in two dimensions. Use at least two buttons that present physics in different media.
- **Project 3: Simulation of projectile motion in two dimensions.** Create a simulation object (button) that has initial vertical velocity and horizontal velocity and that falls under the influence of gravity – moving in two dimensions. Your button should leave a trail.
- **Project 4: Final project.** Students' choice, as long as the project (1) is completed in three days and (2) teaches physics. Students must demonstrate viability of completing the project after the first day. All students chose to create a simulation for their final project.

---

[6]As evidence of their motivation, students never asked what the grading policy was, even when I offered to discuss it with each of them.

Projects were designed to provide increasing complexity both in the programming expertise required and in the kinematics concepts students were asked to deal with. Programming complexity increased from a project involving mostly modification of existing components (Project 1), to a presentation (Project 2, which required few actions), to a simulation which involves calculation and multiple interacting actions. The kinematics concepts students were asked to deal with went from one dimensional projectile motion to two dimensional projectile motion, the standard sequence of increasing complexity in physics' texts [Hewitt, 1989 #370; Serway, 1989 #371].

The purpose of Project 4 was to provide an opportunity for the students to explore a direction in which they were interested. I saw Project 4 as an opportunity for students to demonstrate their interest and engagement with the domains and with using Emile for that exploration. Researchers in Logo and Boxer have noted how students, given the opportunity to set their own design criteria, will often invent creative and powerful representations suggesting that students are learning deep understandings [Harel, 1990 #108; Kafai, 1993 #131; Ploger, 1991 #23]. In the examples of Logo and Boxer, students spend a good deal of time developing their artifacts—time which is necessary to develop this deep learning. While I could not then expect the same level of representations in the Emile-using students in such a short period of time, I saw Project 4 as an opportunity for students to demonstrate their engagement by attempting interesting projects. If the projects were dull or un-interesting, it could be that the students lacked interest in the domain or Emile.

3.2 Data Collected and Analysis Methods

Three kinds of data were collected to address the questions noted at the beginning of this section:

- Log files to determine use and fading of scaffolding.
- Student projects and Design Notebooks to assess performance.
- Clinical interviews were used to assess learning.

3.2.1 Studying use and fading of scaffolding with log files

I evaluated use of Emile through the analysis of *log files* and by review of student Design Notebooks. Log files are recordings of user interactions with software that are created as the student uses Emile. (See Card, Moran, and Newell [Card, 1983 #32] for early descriptions of log files and some examples of use.) Emile's log files record student button clicks, menu selections, window clicks, when text was entered, and other interface actions. Student Design Notebooks provide examples of how students used the scaffolding (e.g., what they wrote for articulations, how they arranged their Project Charts).

Some example log file entries are:

```
8/13/92,10:08:25 AM,Project Description, Chapter Heading, Edited,
Project Description
8/13/92,10:16:20 AM,A Droppable Object, Button, menuSelect,
copyToNotebook
```

The first entry indicates that the user edited the Project Description on the Project Description page, and the second indicates that the user made a menu selection to copy the button "A Droppable Object" to his notebook from the component library.

By reviewing log files for adaptation incidents (i.e., modifications to the Preferences page) and noting which micro activities were selected by students (e.g., creation of a Plan page, positioning an action), I could identify:

- Strategies that students used with Emile,
- Differences in use between students, and
- When students adapted their scaffolding.

3.2.2 Assessing performance with student artifact analysis

To evaluate a student's model-building and programming performance as *successful*, I need to show that (1) the student was able to program and understand the program he created and (2) the student was able to understand the program as a model of a phenomenon in kinematics. I operationally define successful performance of programming for model-building activities with Emile as student creation of:

(1) A working program that achieves project goals through student modification (i.e., simply assembling components as-is from the Library was not success). Studies by Pea and Kurland [Pea, 1986 #146] and those described in Ambron and Hooper [Ambron, 1990 #126] suggest that few students are able to achieve this part of the definition. These studies describe student programmers as understanding little more than how to write correct program code (Pea and Kurland) or not being able to write complex programs (e.g., no longer than five or ten lines) (Ambron and Hooper). To achieve the four projects described previously, students had to construct programs at least forty lines long and demonstrate understanding by modifying those programs.

(2) Articulations that explain the program as representing kinematics content. This part of the definition says that there is kinematics content in creating the projects and that a successful performance of model-building involves recognition of that content.

As an example application of the test, I use student L's two-dimensional projectile motion simulation. Figure 20 is a screenshot from student L's project. The button Positive Gravity can be dragged on the screen, and when released, it will launch with the specified horizontal velocity and starting vertical velocity. Using the definition of successful performance described in this section, student L was successful at this project: He achieved the project goals with a working program on which he made extensive modifications, and he recognized the kinematics content of the project.

- This project successfully meets the project goals: No buttons cause errors when clicked upon, and the Positive Gravity button does launch and fall with action similar to that of a real world projectile in two-dimensions.
- The student not only made the modifications necessary to achieve the minimum goal, but he made others besides. The modifications he made suggests a good understanding of the program. For example, he changed how the path of the falling projectile is traced from dots to a line, which required understanding of the falling behavior enough to understand where and how tracing should occur with lines (which involved recording different kinds of data about the falling projectile than was used previously).
- The student indicated that he understood the physics content of this project through his project evaluation, quoted below. (Statements in bold are prompts on the evaluation form.)

> **What will a student who uses this project learn about Physics?**
> They will learn about horizontal and vertical velocity, and how the curves change according to their velocity.
> **What will they do or see in this project which will help them learn about Physics?**
> They will see a field in which there's charts where the user can set the horizontal and vertical velocity to move the button "Positive Gravity."

3.2.3 Assessing learning with clinical interviews

A clinical interview approach was used to studying student learning with Emile. Clinical interviews as an assessment technique are described in [Finley, 1984 #351; Posner, 1982 #350; Novak, 1984 #349]. Alternative forms for assessing student learning include a multiple-choice test (as used in [White, 1984 #283; Roschelle, 1991 #224]) or other standardized measure. As Posner and Finley both point out, however, only with the interview can the evaluator have any opportunity to note interactions and unexpected results from an instructional intervention. I chose the clinical interview format because of the unusual nature of the intervention.

Students were interviewed at the beginning and end of the workshop. Students were asked to respond to a task (same tasks in pre- and post-interviews), to a set of probes asked of every student on both interviews, and to additional probes based on student responses to explore their understanding of four scenarios – two on physics and two on programming. I conducted all interviews. The style of interview was based on Novak [Novak, 1984 #349] in terms of use of countersuggestions (e.g., students were occasionally queried if they were sure of their response on both correct and incorrect responses) and allowing reference to the pre-interview on the post-interview to invite student comparison of responses (e.g., "Last time I said X, can I just say it again?"). All interviews were videotaped and transcribed to text.

The four problem situations posed to students, with the instructions given to students and the standard probes used by the interviewer, are described below:

- **Question 1: On velocity and acceleration**. Let's say that you left right now for the Baskin-Robbins up on the corner of South and East University (an ice cream store about two blocks from the workshop site) and it takes you two minutes to get there. Using whatever units you might typically use to measure speed, how fast did you travel? Probes include: What is velocity? Did you accelerate? What is acceleration?
- **Question 2: On projectile motion**. You're standing on the edge of the roof of this building and you drop a rock off the roof. Where does it land relative to the edge of the building? Using whatever units you would typically use to measure speed, how fast is it going when it hits? Probes include: Would it matter if you just dropped the rock or if you threw it? What is the rock's vertical and horizontal velocity?
- **Question 3: On reading and writing a mixed media (text and graphics) program.** I start a computer program for you that has the screen like this (a screenshot is provided to the student). What do you think it's for? What do you think it does? How do you think it works? Probes include: What is a button? What is a field? How would you make this button? Would it be hard to change this program?
- **Question 4: On reading a text program**[7]**.** Here's a computer program, the behavior of some button (a program text is provided). Can you tell me what it does? Probes include: What are your clues to tell you what this program does? Where have you seen a line like that before? What does the user see while the program runs?

Interviews were coded using a technique similar to Krajcik's and Magnusson's [Krajcik, 1990 #145; Magnusson, 1991 #374; Magnusson, 1993 #377] which Magnusson refers to as the constant comparative method.. I reviewed the students' responses to the clinical interview and identified student statements about physics and programming concepts. I compared these statements to descriptions of student conceptualizations for physics and programming in the literature. Based on this comparison, I created a categorization scheme which included literature categories but extended them to include categories which I saw reflected in the students' responses.

The three key physics topics for categorization were velocity, acceleration, and projectile motion. I chose these as the key three topics in kinematics [Hewitt, 1989 #370; Serway, 1989 #371]. Literature on student conceptualizations of these topics (e.g., [Arons, 1990 #9; diSessa, 1982 #65; Trowbridge, 1980 #288; Trowbridge, 1981 #287; VanHeuvelen, 1991 #198]) dichotomizes student conceptualizations into two types: Pretheoretical (naive physics) and theoretical (correct and expert-like conceptualizations). Starting from these literature distinctions and informed by the interview statements, a three level categorization emerged of increasing sophistication – level 1 was

[7] I chose not to focus a problem on writing a text program because students were not required to write programs in Emile. One of the probes for question 3 does ask students to describe a text program (just define the components, not write the syntax), and even on that task, students performed poorly.

pretheoretical and level 3 was the most expert-like (see [Guzdial, 1993 #104] for more details on the coding scheme).

Programming concepts were categorized on two levels:

- Student knowledge on *reading* versus *writing* programs. Since Emile activities required reading text language programs but not necessarily write them, such a distinction was important to explore.
- Student knowledge of *text* (traditional) versus *text-and-graphics* (non-traditional, e.g., including buttons and fields, triggers, and behaviors).

4. Results of Evaluation

Below are the results of the evaluation on each of the three evaluation topics:

1. Use and fading of scaffolding
2. Performance
3. Learning

4.1 Results on use and fading of scaffolding

Students differed dramatically in their use of Emile's scaffolding and in when they faded their scaffolding. Program three (the creation of a two-dimensional projectile motion simulation) is a good place to characterize differences between students. All students had completed two programs and were developing individual styles. This was the second simulation they had built, so they were familiar with the format and the tools available to them. This section will contrast students C and B to highlight the diversity in terms of use of the scaffolding, even though each had the same Preferences at the beginning of program 3. On the day that the students began working on program 3, I had given a presentation such that they only had one hour of the three hour session to begin work.

Student C made extensive use of the library with frequent testing as he began his program three:

- First, he copied both the *Droppable Object* button and the *Gravity Simulation* group into his Design Notebook from the Library. His explanation at the time was that he wasn't sure which he'd need, so he thought he would take both.
- He composed the entire *Gravity Simulation* group, tailored his Project Chart (i.e., organized the icons in a structure he preferred), and tested his project.
- He then emptied the slot for the initial velocity in the simulation object (*Positive Gravity* button) and selected the menu item to fill the slot, but canceled out of that action.
- He created a field, named it *Starting Velocity*, and linked it to the *Gravity Simulation* group.
- He then returned to the *Positive Gravity* button, again asked to fill the slot, and selected the field *Starting Velocity* to fill the initial velocity slot.
- He tested his project again with different values for the initial velocity entered into the *Starting Velocity* field.
- He returned to the Library and copied actions *Save Left of Button*, *Set Left of Button*, and *Increment Value* (which had been discussed as being useful in implementing horizontal motion).
- He composed all three actions into the button *Positive Gravity*.
- He ended the day with making a Journal page and typing an entry:

  > Today I started to create my button which will move up and have horizontal pull at the same time.

Student C's completed project is figure 22.

Student B began with articulation and more creation of new objects than copying of Library objects:

- Student B began by entering a Project Description.

  > This project will be to create a button to launch from any particular place on the screen that the user defines. The user can also select the velocity (both vertical and horizontal). This will help people to understand the strange and crazy laws of physics. So I hope you like it.

- He copied the button *Droppable Object* to his Notebook and tailored the appearance of his Project Chart.
- He returned to the library and coped several actions to his Notebook: *Set Left of Button*, *Save Left of Button*, *Sound a Beep*, and *Clear the Graphics*.
- He created a new button and named it *Clear Screen*.
- He composed the actions *Clear the Graphics* and *Sound a Beep* into the new button, then tested it five times.
- He created a new field and named it *Time in Motion*.
- He created two new goals and named them *Initial Vertical Velocity* and *Initial Horizontal Velocity*.
- He created two new fields and named them also *Initial Vertical Velocity* and *Initial Horizontal Velocity*.
- He composed the button *Droppable Object* and tested it, but received an error because some of the fields which *Droppable Object* required (e.g., the field *Time*) were not created.
- He made a Journal entry:

  > Today was an OK day in terms of productivity, but I didn't do that much. This is a new project called Launching Things. It is going to work but I don't know when.

Student B's completed project is figure 23. It looks different than students' C or L's program because B decided to separate the launching function from the object to be launched. In B's version of the two-dimensional projectile motion simulation, the user moves the *Launching Thing* (what he renamed *Droppable Object* to be) and clicks *Launch It* to begin the simulation. The distinction is notable because, in order to separate this functionality, student B had to recreate all the functionality of the original *Droppable Object* in the button *Launch It*.

These two sequences of actions exemplify how students used Emile's scaffolding:

- *Communicating Process*: Students made frequent use of most of the components in the structure presented to the students by Emile's scaffolding. Groups and goals were not used frequently (i.e., the majority of student-created objects belonged to no groups, and less than one group per student per project had a matching goal), except for copying large groups of objects from the library. However, all other elements (buttons, fields, actions) were used frequently – even as scaffolding faded.

  Table 6 notes in which projects students first turned off the scaffolding Preferences associated with actions and slots. Note that different students turned off various scaffolding at various times: Student S faded his scaffolding early on, student M soon after, and students C and L never created actions nor directly edited their behaviors. Not shown in this table is that all students (including S) turned the scaffolding back on occasionally, even in the fourth program, to go back to actions and slots during difficult parts of their programs.

- *Coaching:* Notice that neither student in the above examples referred to Design Stage pages or were careful about using goals and groups before defining lower-level components. Coaching scaffolding was faded by all students by the second program, though not all with positive results. Students C and L frequently requested help in their process after removing all the coaching scaffolding. Perhaps they would have understood their process better had they left the scaffolding on for longer.

- *Eliciting Articulation:* As seen in the above examples, prompts were infrequently accessed by students, e.g., predictions were not used in the above examples and rarely over all. Journals were used often due to instructor's prodding, but in general, the macro elicitation of scaffolding was not wildly successful. Instead, students tended to talk to one another, showing off their programs, and discussing their programs' validity. I suspect this social articulation and reflection (1) alleviated the need for prompted articulation and reflection and (2) had a greater perceived benefit for the students than writing notes to themselves in their Design Notebooks.

  However, the above examples do suggest the importance of the micro level eliciting articulation scaffolding. Students chose names for their objects which corresponded well with the physics concepts they were exploring. These names were important in helping the students understand the computational process of their programs and in assembling the programs to correspond to their physics understandings – two of the critical problems in programming identified earlier.

4.2 Results on performance

Table 7 summarizes the performance of the five students on each of the four projects in terms of programming performance (e.g., constructing a working program that achieved the project goals through student modifications) and model-building performance[8] (e.g., demonstrating an understanding of the program as a model of kinematics). Check marks indicate successful performance, and a dash indicates partially or not successful performance.

Students were remarkably successful at both programming and model-building activities, given the problems from the literature described earlier. Students using Emile did construct several interesting programs (e.g., non-trivial with physics content), and their articulations indicate that they saw their programs as simulations of kinematics concepts. For example, note the projects by students L (figure 21), C (figure 22), and B (figure 23). All three of these were successful simulations of two-dimensional projectile motion, completed by students in nine hours (after completing two other programs), with no previous programming experience before beginning the class.

As mentioned earlier, the students' final projects were self-directed, to see whether students did choose interesting projects. As seen by their choices, they attempted some sophisticated programs:

- L created a game where the user chooses initial horizontal and vertical velocity to launch a golf ball over a lake.
- B created a game in which a plane moved across the top of the screen and released a jumper. The user chooses the point at which the jumper's parachute opens. The parachute's opening changes the acceleration for the jumper.
- C created a bouncing object which repeatedly freely falls then bounces up with reduced velocity on each bounce.
- M extended Droppable Objects by contrasting an erroneous use of instantaneous velocity to calculate the simulation object's position (which was used in the Library simulation objects to simplify the computations) with a more accurate method using average velocity.
- S extended Launchable Objects by correcting the position calculation (as M had) and adding effects due to wind (i.e., taking into account "resistance.")

These results are particularly indicative of student engagement in their use of Emile. Students could have created relatively simple final programs, but all of them chose sophisticated and

[8]Since Project 2 was not a simulation, physics performance is rated here in terms of accuracy of physics concepts.

interesting objectives. On the last day of class, I offered students the opportunity to take their final programs home with them (which was possible because the final product could be run outside of Emile). I expected that students would want to take their final programs home with them because of their interest in those programs. Instead, all the students asked to take all of their programs home with them, and several of the students asked if they could have copies of Emile as well. This anecdotal evidence suggests that students were interested in what they were doing with Emile.

4.3 Results on learning

This section presents the results on student learning about programming and physics as evidenced by their comments in clinical interviews. Table 8 summarizes the change in physics conceptualizations (learning) for the five students between the pre-interview[9] and post-interview. In general, all students learned about physics during their activity with Emile, which is a striking result given the learning difficulties that students experience as described in the literature on physics learning (e.g., [Arons, 1990 #9; Trowbridge, 1980 #288]).

Student B is a particularly good example of the robust physics understanding exhibited in the post-interviews.

- *Pre-interview conceptualization of projectile motion.* Student B's comments in response to the second problem (about dropping or throwing a rock off the roof of a building) suggest a level 2 conceptualization of projectile motion – he did not have traditional pretheoretical conceptualization (e.g., that the weight of an object determines the speed of a falling object), but nor could he explain time and velocity of a falling object. In the quote below, he gives up trying to compute the velocity of the rock falling.

  R: [Question 2 on where a rock lands that is dropped from the roof and how fast it's going when it hits]
  B: Where does it land? Beneath me, below where my hand is, unless there's wind. And the velocity is, let's see, gravity...umm...I can't remember that

- *Post-interview conceptualization of projectile motion:* Student B's responses to problem two in the post-interview suggest a sophisticated (level 3) conceptualization of projectile motion. While he does make mistakes in his calculations, he recovers from them and demonstrates that he understands the relationships between horizontal and vertical velocity and even instantaneous and average velocity. His understanding of the interaction between acceleration, velocity, and position at a detailed level during a projectile's flight is particularly interesting. He has progressed from simply recognizing that acceleration increases vertical velocity (as he stated in his pre-interview, a statement indicating a level 2 conceptualization), to being able to simulate on a second-by-second basis what he thinks is going on with the projectile.

  R: Can you tell me how long it took the rock to get to the ground?
  B: It would be about one second
  R: Okay, where did you get that from?
  B: If the acceleration is 30 feet per second per second, then per second it will be going 30 feet per second, then it will just take a little longer for it to get to the ground.
  R: Why?
  B: Because you have to divide the, to get the average velocity, which is how fast it's going, and how you can measure how far it's gone, you have to...let's see...it will be going, it will be going 15 meters per second. Maybe two seconds, I guess.
  R: Why?
  B: Because...1.5 seconds. Because, by the time it's accelerated the second second, it will be going about 45 feet per second, so it'll have to be between the first and second second that it hits the ground.

[9]Illness prevented student M from participating in the pre-interview.

R: In the second second, it's going 45 feet per second?
B: Yeah, I think so.
R: Where did you get that number from?
B: Umm, adding the 30 meters per second, you add that to the 45, and you have to divide it by 2. Oh yeah, I forgot about that part. So that'll be 22.5 feet per second. 22.5 meters, I mean, feet per second. Then it'll probably be 2 and a half seconds.

Students learning about programming was similarly positive, but not as powerful. On the pre-interview, no student was able to explain how to read or write a HyperTalk program and no student was able to explain how to write a graphics-and-text program. All students were able to interpret the buttons and fields shown to them and explain what they did.

On the post-interview, students were able to describe how to write graphics-and-text programs using the metaphors of Emile (e.g., buttons, triggers, slots), but only three students were able to read a text program (students B, M, and S) and *none were* able to write one. This last result was somewhat surprising since several of the students were directly typing text programs in the last program. However, even student S, the first student to begin typing programs directly, referred to actions instead of text code when asked to explain how to build a text program:

R: Do you know what it takes to make these buttons do what they do? If I gave you pencil and paper, could you write it down?
S: No, I don't think that I know, well...I don't know. I might be able to figure it out with HyperCard – that's the only Mac language I know.
R: Well, assume that this is Emile. Do you think you could do it?
S: Yeah! Go to the library, find the actions Play a Sound and Record. That's all there is – play and record. And you'd need to load the picture.

Student C is a good example of a student who created several interesting programs and developed confidence in his programming ability, though that ability was dependent on the scaffolding in Emile. C had no previous programming experience before this class.

R: How hard would it be to change what these things do?
C: Not hard at all. You go to click on it, and it'll show you what it took to create it all. Then go find what you want to change. If zero was two, or it's 10 and you want to make it 20. Just click on it and change it.
R: Could you write down for me what each of these behaviors look like?
C: Not without looking at it
[…]
R: Let's say that you're building something new. Let's say that you're building one of these things. What would you do first?
C: I go copy all these down from the main library...Then I could look at them, and if I needed them, they'd be there. Then I'd just read the whole thing in and see how it works....Then I'd go to the Project Chart – it's probably got to have one of them like we do – to see what all links together to cause it to make it to work.

5. Discussion and Conclusions

The results indicate that Emile was quite successful in this study.

- Students did use Emile's scaffolding and changed that scaffolding based on individual needs and interests.
- Students were successful at programming interesting models of physics.
- They learned about programming and new conceptualizations of physics.
  - What students learned about programming was the process being communicated by Emile's scaffolding (e.g., actions, slots, fields, and so on.) That they did not learn more

(e.g., the underlying text programming language) is acceptable since learning traditional programming was not the goal. This result suggests that Emile was successful at facilitating learning *about* programming.

– Students learned new ways of looking at velocity, acceleration, and projectile motion. This suggests that Emile was successful at facilitating learning *through* programming – that is, using programming as leverage to learning another domain (kinematics).

After declaring Emile a successful environment for teaching physics through programming, one next question is whether Emile is more successful than other kinds of interventions—that is, whether it is an efficient way to teach physics. I think that the question is, in at least one sense, unimportant because of the well-documented difficulty of getting any kinematics learning at all among high school students—even with comparable hours spent on the subject [Roschelle, 1991 #224; Arons, 1990 #9; Champagne, 1985 #329]. Nonetheless, it may be that the students' learning documented here may be realizable with less time—that is, students may have gained the same concepts with fewer projects. On the other hand, students in a more traditional physics class do not have access to the same computational resources as these students in the same large blocks of time. It is an interesting challenge to explore just how efficient physics learning through programming could become.

While Emile is just one example of software-realized scaffolding, we can use the experience as an indication of what we might expect from software-realized scaffolding in other contexts. Of the three kinds of scaffolding, it seems that communicating process in Emile was the most successful in terms of student reference to the relevant facilities in interviews, the frequency of use, and least fading to the lowest levels. Coaching and eliciting articulation were less successful on these same grounds. This may not be an indication of the relative importance of these scaffolding forms overall but may have been hindered by the implementations in Emile. Coaching in Emile was not state of the art, as compared with Anderson [Anderson, 1989 #479; Anderson, 1990 #515] or Fischer [Fischer, 1987 #84; Fischer, 1990 #319]. Further, articulating may be elicited more effectively if the articulations were used to some purpose, such as in a collaborative learning environment (e.g., [Scardamalia, 1991 #53; Scardamalia, 1989 #54]).

The fading of scaffolding was quite successful. Students did use Emile at many different levels of scaffolding and developed a wide range of strategies for use of the scaffolding. Nevertheless, there were definite flaws.

- First, there was no real structure to the fading of scaffolding. Preferences were not ordered in any way which was meaningful to the students, and students were given little support on when and how to fade scaffolding. Advising on scaffolding would be a useful addition to any future implementation of adaptable software-realized scaffolding.
- Second, and more importantly, there was no coach or critic to analyze how students manipulated the scaffolding to insure that they did not hurt their performance or learning. Students do not typically have good metacognitive skills [Brown, 1983 #27]. Choosing one's scaffolding level is clearly a metacognitive activity. While it's a good strategy for students to explore their ability by reducing their scaffolding, there needs to be some safety measure to inform the student when the experiment is unsuccessful – a net to catch them if they fall. Such a net was not implemented in Emile.

While an interpretation of educational software features as scaffolding is not a panacea which explains all innovation in educational software or serves as a complete guideline for design of new educational software, software-realized scaffolding presents an important perspective which can provide new explanatory and design leverage. Teaching is not a new activity, and important pedagogical principles have been identified over the millennia. The approach of software-realized scaffolding is to identify an important set of teaching activities and to implement them (as much as possible) in software.

- As an explanatory tool, software-realized scaffolding can be used to classify features of software and perhaps identify where some components of software-realized scaffolding were missing.
- As a design tool, software-realized scaffolding provides a list of features which are important to provide in order to facilitate learning through and about student activity.

Acknowledgments

The research on Emile was supported by National Science Foundation grant #MDR-9010362 and by Apple Computer. Appreciation for comments and direction on this work go to the anonymous reviewers, Patricia Baggett, Carl Berger, Phyllis Blumenfeld, Brian Schunck, and especially, Elliot Soloway.

References

1. Adelson B., Soloway E. (1984) “A cognitive model of software design.” Yale University. Cognition and Programming Project
2. Ambron S., Hooper K., eds. (1990)*Learning with Interactive Multimedia: Developing and Using Multimedia Tools in Education*. Redmond, WA: Microsoft Press:
3. Anderson J.R., Boyle C.F., Corbett A.T., Lewis M.W. (1990) “Cognitive modeling intelligent tutoring.” *Artificial Intelligence*;42:7-49.
4. Anderson J.R., Conrad F.G., Corbett A.T. (1989) “Skill acquisition and the LISP tutor.” *Cognitive Science*;13:467-505.
5. Arons A.B. (1990) *A Guide to Introductory Physics Teaching*.New York: John Wiley & Sons
6. Blumenfeld P.C., Soloway E., Marx R.W., Krajcik J.S., Guzdial M., Palincsar A. (1991) “Motivating project-based learning: Sustaining the doing, supporting the learning.” *Educational Psychologist*;26(3 & 4):369-398.
7. Brasell H. (1987) “The effect of real-time laboratory graphing on learning graphic representations of distance and velocity.” *Journal of Research in Science Teaching*;24(4):385-395
8. Brown A.L., Bransford J.D., Ferrara R.A., Campione J.C. (1983) “Learning, remembering, and understanding.” In: Kessen W, ed. *Handbook of Child Psychology: Cognitive Development*. New York, NY: Wiley: vol 3).
9. Brunner C., Hawkins J., Mann F., Moeller B. (1990) “Designing Inquire.” In: Bowen B, ed. *Design for Learning*. Cupertino, CA: Apple Computer
10. Card S.K., Moran T.P., Newell A. (1983) *The Pyschology of Human-Computer Interaction*.Hillsdale, NJ: Lawrence Erlbaum and Associates
11. Champagne A.B., Gunstone R.F., Klopfer L.E. (1985) “Effecting changes in cognitive structures among physics students.” In: West LHT, Pines AL, ed. *Cognitive Structure and Conceptual Change*. Orlando, FL: Academic Press: 163-188.
12. Collins A. (1990) “Cognitive apprenticeship and instructional technology.” In: Jones BF, Idol L, ed. *Dimensions of Thinking and Cognitive Instruction*. Hillsdale, NJ: Erlbaum and Associates:
13. Collins A., Brown J.S. (1988) “The computer as a tool for learning through reflection.” In: Mandl H, Lesgold A, ed. *Learning Issues for Intelligent Tutoring Systems*. New York: Springer:
14. Collins A., Brown J.S., Newman S.E. (1989) “Cognitive apprenticeship: Teaching the craft of reading, writing, and mathematics.” In: Resnick LB, ed. *Knowing, Learning, and Instruction: Essays in Honor of Robert Glaser*. Hillsdale, NJ: Lawrence Erlbaum and Associates:
15. Decker R.W., Hirshfield S.H. (1990) “A survey course in computer science using HyperCard.” *SIGCSE Bulletin*;22(1):229-235.
16. diSessa A. (1982) “Unlearning Aristotelian physics: A study of knowledge-based learning.” *Cognitive Science*;6:37-75.
17. diSessa A. (1985) “A principled design for an integrated computational environment.” *Human-Computer Interaction*;1(1):1-47.

18. diSessa A. (1991) “Local sciences: Viewing the design of human-computer systems as cognitive science.” In: Carroll J, ed. *Designing Interaction: Psychology at the Human-Computer Interface*. New York: Cambridge University Press
19. diSessa A.A., Abelson H. (1986) “Boxer: A reconstructible computational medium.” *Communications of the ACM*;29(9):859-868.
20. diSessa A.A., Abelson H., Ploger D. (1991) “An overview of Boxer.” *TheJournal of Mathematical Behavior*;10(1):3-15.
21. DuBoulay B., O'Shea T., Monk J. (1989) “The black box inside the glass box: Presenting computing concepts to novices.” In: Soloway E, Spohrer JC, ed. *Studying the novice programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates.
22. Earnshaw R.A., Wiseman N. (1992) *An Introductory Guide to Scientific Visualization*.Berlin: Springer-Verlag
23. Eylon B.-S., Linn M.C. (1988) “Learning and instruction: An examination of four research perspectives in science education.” *Review of Educational Research*;58(3):251-301.
24. Farnham-Diggory S. (1990) *Schooling*.Cambridge, MA: Harvard University Press(Bruner J, Cole M, Lloyd B, eds. The Developing Child
25. Finley F.N. (1984) “Using propositions from clinical interviews as variables to compare student knowledge.” *Journal of Research in Science Teaching*;21(8):809-818.
26. Fischer G., Burton R.R., Brown J.S. (1978) “Aspects of a theory of simplification, debugging, and coaching.” BBN Labs. Technical Report #3912. July.
27. Fischer G., Lemke A.C. (1987) “Construction kits and design environments: Steps toward human problem-domain comunication.” *Human-Computer Interaction*;3:179-222.
28. Fischer G., Lemke A.C., Mastaglio T., Morch A. (1990) “Using critics to empower users.” In: *Human Factors in Computing Systems, CHI'90 Conference Proceedings (Seattle, WA)*. New York: ACM: 337-347.
29. Fischer G., Lemke A.C., Mastaglio T., Morch A.I. (1991) “The role of critiquing in cooperative problem solving.” *ACM Transactions on Information Systems*;9(3):123-151.
30. Garlan D., Miller P. (1984) “Gnome: An introductory programming environment based on a family of structure editors.” In: *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*. New York: ACM-SIGSOFT/SIGPLAN
31. Goodman D. (1977) “The two faces of HyperCard.” *MacWorld*;:122-129.
32. Guzdial M., Soloway E., Blumenfeld P., et al. (1992) “The future of CAD: Technological support for kids building artifacts.” In: Balestri D, Ehrmann S, Ferguson DL, ed. *Learning to design, designing to learn: Using technology to transform the curriculum*. Norwood, NJ: Ablex Publishing Company.
33. Guzdial M.J. (1991) “The need for education and technology: Examples from the GPCeditor.” In: *Proceedings of the National Educational Computing Conference*. Phoenix, AZ: 16-23.
34. Guzdial M.J. (1993)*Emile: Software-realized scaffolding for science learners programming in mixed media* [Unpublished Ph.D. dissertation]. University of Michigan.
35. Halloun I.A., Hestenes D. (1987) “Modeling instruction in mechanics.” *American Journal of Physics*;55(5):455-462.
36. Harel I., Papert S. (1990) “Software design as a learning environment.” *Interactive Learning Environments*;1(1):1-32.
37. Hestenes D. (1987) “Toward a modeling theory of physics instruction.” *American Journal of Physics*;55(5):440-454.
38. Hestenes D. (1992) “Modeling games in the Newtonian world.” *American Journal of Physics*;60(8):732-748.
39. Hewitt P.G. (1989) *Conceptual Physics*. (Sixth ed.) Glenview, Ill: Scott, Foresman and Company
40. Hohmann L., Guzdial M., Soloway E. (1992) “SODA: A computer-aided design environment for the doing and learning of software design.” In: *Computer assisted learning: 4th international conference, ICCAL '92 proceedings*. Berlin: Springer-Verlag: 307-319.
41. Jeffries R., Turner A.A., Polson P.G., Atwood M.E. (1981) “The processes involved in designing software.” In: Anderson JR, ed. *Cognitive Skills and Their Acquisition*. Hillsdale, NJ: Lawrence Erlbaum Associates

42. Johnson W.L., Soloway E. (1985) “PROUST: An automatic debugger for Pascal programs.” *BYTE*;10(4):179-190.
43. Kafai Y.B. (1993)*Minds in Play: Computer Game Design as a Context for Children's Learning* [Unpublished Ph.D. Dissertation]. Graduate School of Education of Harvard University.
44. Kay A.C. (1993) “The early history of Smalltalk.” In: Sammet JE, ed. *History of Programming Languages (HOPL-II)*. New York: ACM: 69-95.
45. Krajcik J.S., Layman J.W. (1990) “Middle school teachers' conceptions of heat and temperature: Personal and teaching knowledge.” Paper presented at the meeting of the National Association for Research in Science Teaching meeting, San Francisco, CA.
46. Lave J. (1993) *Tailored Learning: Education and Everyday Practice among Craftsmen in West Africa*.In Preparation.
47. Lee G. (1993) *Object-Oriented GUI Application Development*.Englewood Cliffs, NJ: PTR Prentice-Hall
48. Lehrer R. (1992) “Authors of knowledge: Patterns of hypermedia design.” In: Lajoie S, Derry S, ed. *Computers as Cognitive Tools*. Hillsdale, NJ: Lawrence Erlbaum Associates:
49. Magnusson S. (1991)*The Relationship between Teachers' Content and Pedagogical Content Knowledge and Students' Content Knowledge of Heat Energy and Temperature* [Unpublished Ph.D. dissertation]. University of Maryland.
50. Magnusson S. (1993) “Approaches to interpretive research.” Personal communication, March.
51. Merrill D.C., Reiser B.J. (1993)*Scaffolding the acquisition of complex skills with reasoning-congruent learning environments*. Workshop in Graphical Representations, Reasoning and Communication from the World Conference on Artificial Intelligence in Education (AI-ED'93). The University of Edinburgh: : 9-16.
52. Merrill D.C., Reiser B.J., Beekelaar R., Hamid A. (1992) “Making processes visible: Scaffolding learning with reasoning-congruent representations.” In: *Intelligence Tutoring Systems: Second International Conference, ITS'92*. New York: Springer-Verlag: 103-110.
53. Nielsen J., Frehr I., Nymand H.O. (1991) “The learnability of HyperCard as an object-oriented programming system.” *Behaviour & Information Technology*;10(2):111-120.
54. Norman D.A. (1993) *Things that Make Us Smart: Defending Human Attributes in the Age of the Machine*.Reading, MA: Addison-Wesley
55. Novak J.D., Gowin D.B. (1984) *Learning How to Learn*.Cambridge: Cambridge University Press
56. Oren T. (1990) “Designing a new medium.” In: Laurel B, ed. *The Art of Human-Computer Interface Design*. Addison-Wesley: 467-479.
57. Palincsar A.S. (1986) “The role of dialogue in providing scaffolded instruction.” *Educational Psychologist*;21(1-2):73-98.
58. Palumbo D.B. (1990) “Programming language/problem-solving research: A review of relevant issues.” *Review of Educational Research*;60(1):65-89.
59. Papert S. (1980) *Mindstorms: Children, computers, and powerful ideas*.New York, NY: Basic Books
60. Parnas D. (1972) “On the criteria to be used in decomposing systems into modules.” *Communications of the ACM*;15(2):1053-1058.
61. Pea R.D., Kurland D.M. (1986) “On the cognitive effects of learning computer programming.” In: Pea RD, Sheingold K, ed. *Mirrors of Minds*. Norwood, NJ: Ablex Publishing
62. Perkins D.N., Martin F., Farady M. (1986) “Loci of difficulty in learning to program.” Educational Technology Center. Technical Report June.
63. Petre M., Green T.R.G. (1990)*Where to draw the line with text: Some claims by logic designers about graphics in notation*. INTERACT'90. Cambridge, England, 27-31 August.
64. Petre M., Green T.R.G. (1993) “Learning to read graphics: Some evidence that 'seeing' an information display is an acquired skill.” *Journal of Visual Languages and Computing*;4:55-70.
65. Pidd M., ed. (1989)*Computer Modelling for Discrete Simulation*. Chichester, England: John Wiley & Sons, Ltd.
66. Ploger D. (1991) “Learning about the genetic code via programming: Representing the process of translation.” *The Journal of Mathematical Behavior*;10(1):55-77.
67. Polya G. (1957) *How to Solve It*.Princeton, NJ: Princeton University Press
68. Posner G.J., Gertzog W.A. (1982) “The clinical interview and the measurement of cognitive change.” *Science Education*;66(2):195-209.

69. Richards J., Barowy W., Levin D. (1992) “Computer simulations in the science classroom.” *Journal of Science Education and Technology*;1(1):67-79.
70. Riel M.M., Levin J.A., Miller-Souviney B. (1987) “Learning with interactive media: Dynamic support for students and teachers.” In: Lawler RW, Yazdani M, ed. *Artificial Intelligence and Education*. Norwood, NJ: Ablex: 117-134. vol 1).
71. Rogoff B. (1990) *Apprenticeship in thinking: Cognitive development in social context*.New York: Oxford University Press
72. Roschelle J.M. (1991)*Students' Construction of Qualitative Physics Knowledge: Learning about Velocity and Acceleration in a Computer Microworld* [Unpublished Ph.D. Dissertation]. University of California at Berkeley.
73. Scardamalia M., Bereiter C. (1991) “Higher levels of agency for children in knowledge building: A challenge for the design of new knowledge media.” *Journal of the Learning Sciences*;1(1):37-68.
74. Scardamalia M., Bereiter C., McLean R., Swallow J., Woodruff E. (1989) “Computer-supported intentional learning environments.” *Journal of Educational Computing Research*;5(1):51-68.
75. Scardamalia M., Bereiter C., Steinbach R. (1984) “Teachability of reflective processes in written composition.” *Cognitive Science*;8:173-190.
76. Schneider-Hufschmidt M., Kühme T., Malinowski U., eds. (1993)*Adaptive User interfaces: Principles and Practice*. Amsterdam: North-Holland
77. Shneiderman B. (1977) “Teaching programming: A spiral approach to syntax and semantics.” *Computers and Education*;1:193-197.
78. Schon D.A. (1982) *The Reflective Practitioner: How Professionals Think In Action*.New York: Basic Books
79. Serway R.A., Faughn J.S. (1989) *College Physics*. (Second ed.) Philadelphia: Saunders College Publishing
80. Solomon C. (1986) *Computer environments for children: A reflection on theories of learning and education*.Cambridge, Mass.: The MIT Press
81. Soloway E. (1986) “Learning to program = learning to construct mechanisms and explanations.” *Communications of the ACM*;29(9):850-858.
82. Soloway E. (1993) “Should we teach students to program?” *Communications of the ACM*;36(10):21-24.
83. Soloway E., Ehrlich K., Bonar J., Greenspan J. (1982) “What do novices know about programming?” In: Badre A, Shneiderman B, ed. *Directions in Human-Computer Interaction*. Norwood, NJ: Ablex Publishing
84. Soloway E., Guzdial M., Brade K., et al. (1993) “Technological support for the learning and doing of design.” In: Jones M, Winne PH, ed. *Foundations and frontiers of adaptive learning environments*. New York: Springer-Verlag
85. Spohrer J.C. (1989)*MARCEL: A Generate-Test-and-Debug (GTD) Impasse/Repair Model of Student Programmers* [Unpublished Ph.D. dissertation]. Yale University.
86. Spohrer J.C., Soloway E. (1985) “Putting it all together is hard for novice programmers.” In: *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*.
87. Suikaviriya P., Foley J. (1993) “Supporting adaptive interfaces in a knowledge-based user interface environment.” In: *Proceedings of the Intelligent Interfaces Workshop*. 107-114.
88. Tinker R.F. (1990) “Teaching theory building.” In: Technical Education Research Centers, Cambridge, MA
89. Trowbridge D.E., McDermott L.C. (1980) “Investigations of student understanding of the concept of velocity in one dimension.” *American Journal of Physics*;48(12):1020-1028.
90. Trowbridge D.E., McDermott L.C. (1981) “Investigation of student understanding of the concept of acceleration in one dimension.” *American Journal of Physics*;49(3):242-253.
91. Turkle S., Papert S. (1991) “Epistemological pluralism and the revaluation of the concrete.” In: Harel I, Papert S, ed. *Constructionism*. Norwood, NJ: Ablex: 161-192.
92. VanHeuvelen A. (1991) “Learning to think like a physicist: A review of research-based instructional strategies.” *American Journal of Physics*;59(10):891-897.
93. Vaubel K.P., Gettys C.F. (1990) “Inferring user expertise for adaptive interfaces.” *Human Computer Interaction*;5:95-117.

94. White B.Y. (1984) “Designing computer games to help physics students understand Newton's laws of motion.” *Cognition and Instruction*;1(1):69-108.
95. Wood D., Bruner J.S., Ross G. (1975) “The role of tutoring in problem-solving.” *Journal of Child Psychology and Psychiatry*;17:89-100.

	Communicating Process	*Coaching*	*Eliciting Articulation*
Macro Level	Structures and presents the stages of activities in the process.	Reminds students of the overall structure of their process while they engage in the activity.	Encourages students to be explicit about and reflect on their overall process.
Micro Level	Structures and presents the individual activities of the process.	Reminds students of the attributes of the individual activities to be performed (e.g., when are they to be used, how are they to be used).	Encourages students to articulate the role of the individual activities in their process.

Table 1: Scaffolding at both the macro and micro levels

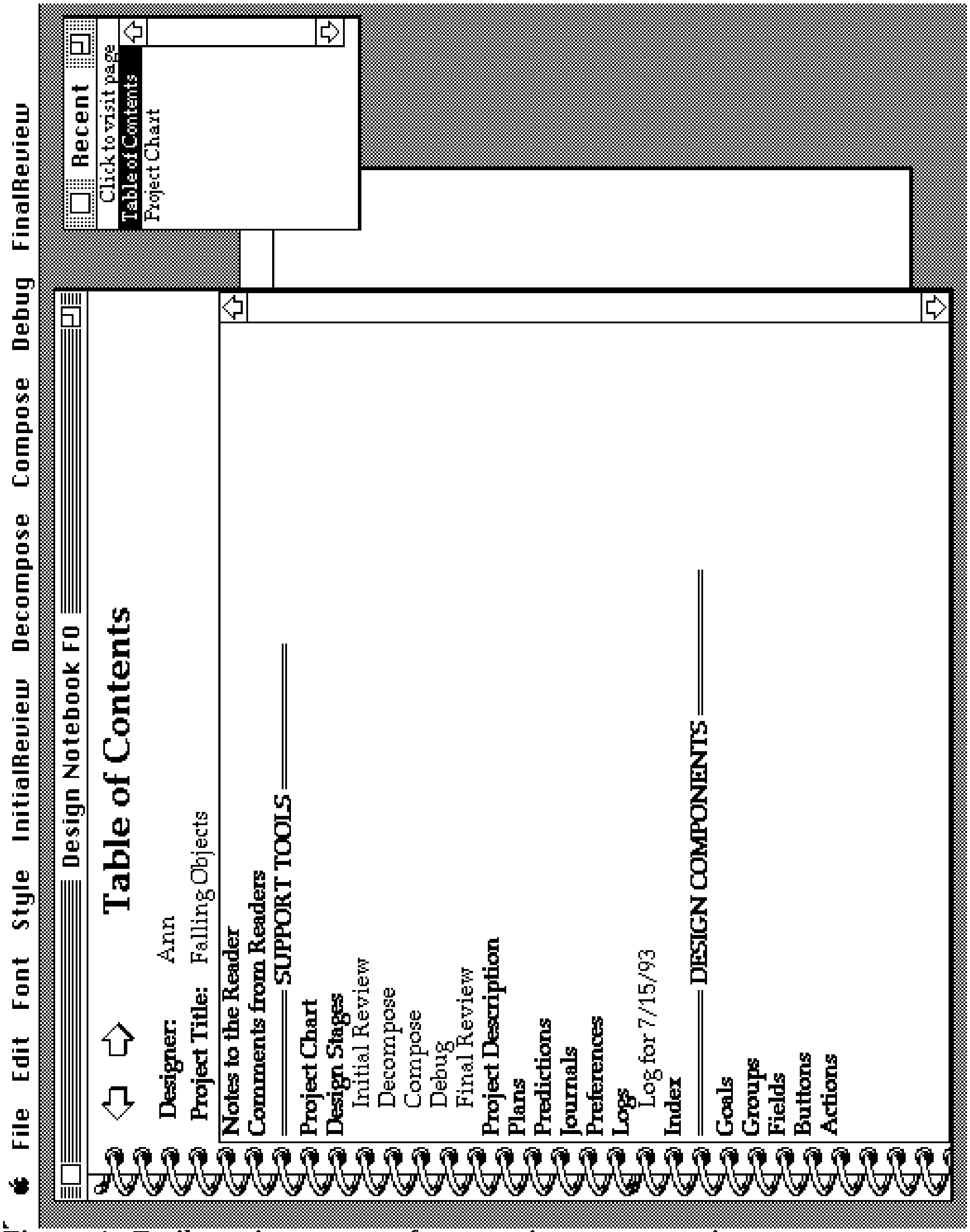

Figure 1: Emile as it appears after creating a new project

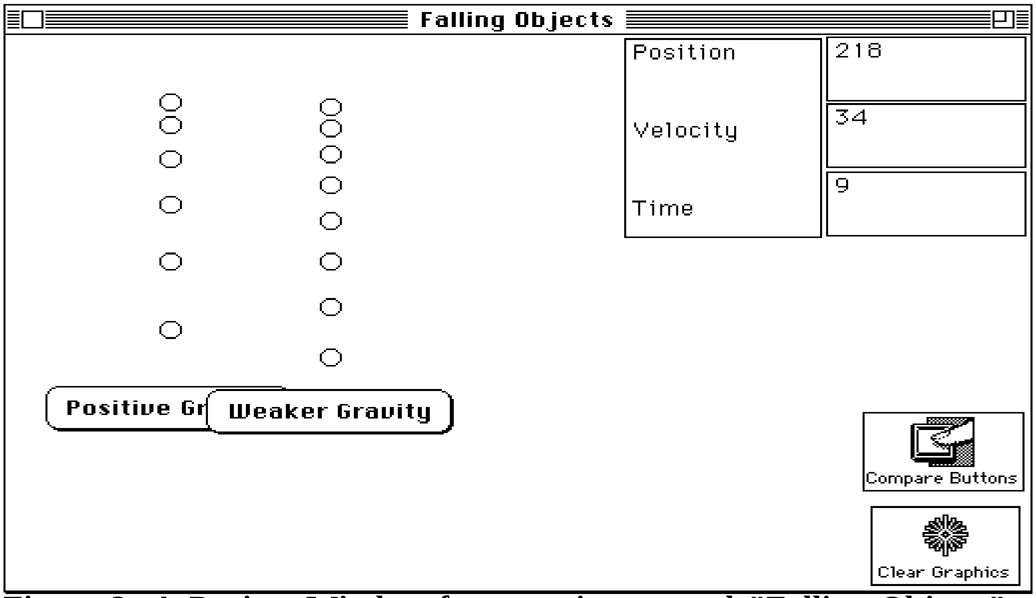

Figure 2: A Project Window for a project named "Falling Objects"

	Macro level	**Micro level**
Communicating Process ***Structuring/Simplifying***	• Design stages	• Actions and slots • Goals and groups
Presenting	• Design Stage pages • Menu names	• Menu items • Library • Representations • Design Notebook
Coaching	• Stage prompting	• Top-down design enforcement
Eliciting Articulation	• Project Descriptions • Plans • Goals • Predictions • Journals	• Naming

Table 2: Software-Realized Scaffolding in Emile

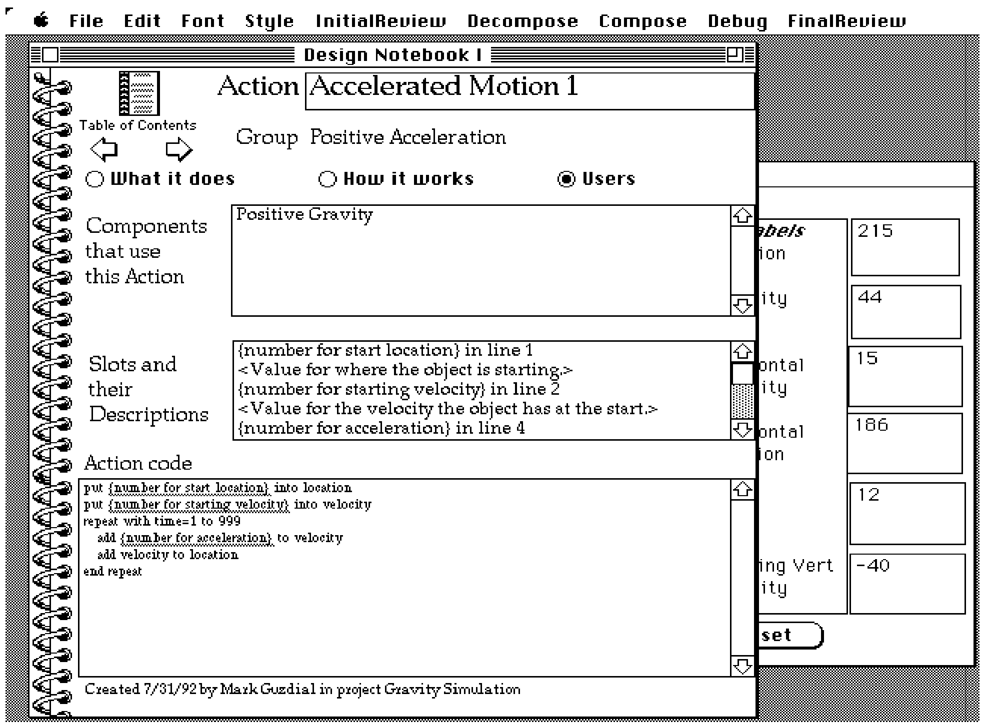

Figure 3: An action which implements the computation of position and velocity for accelerated motion

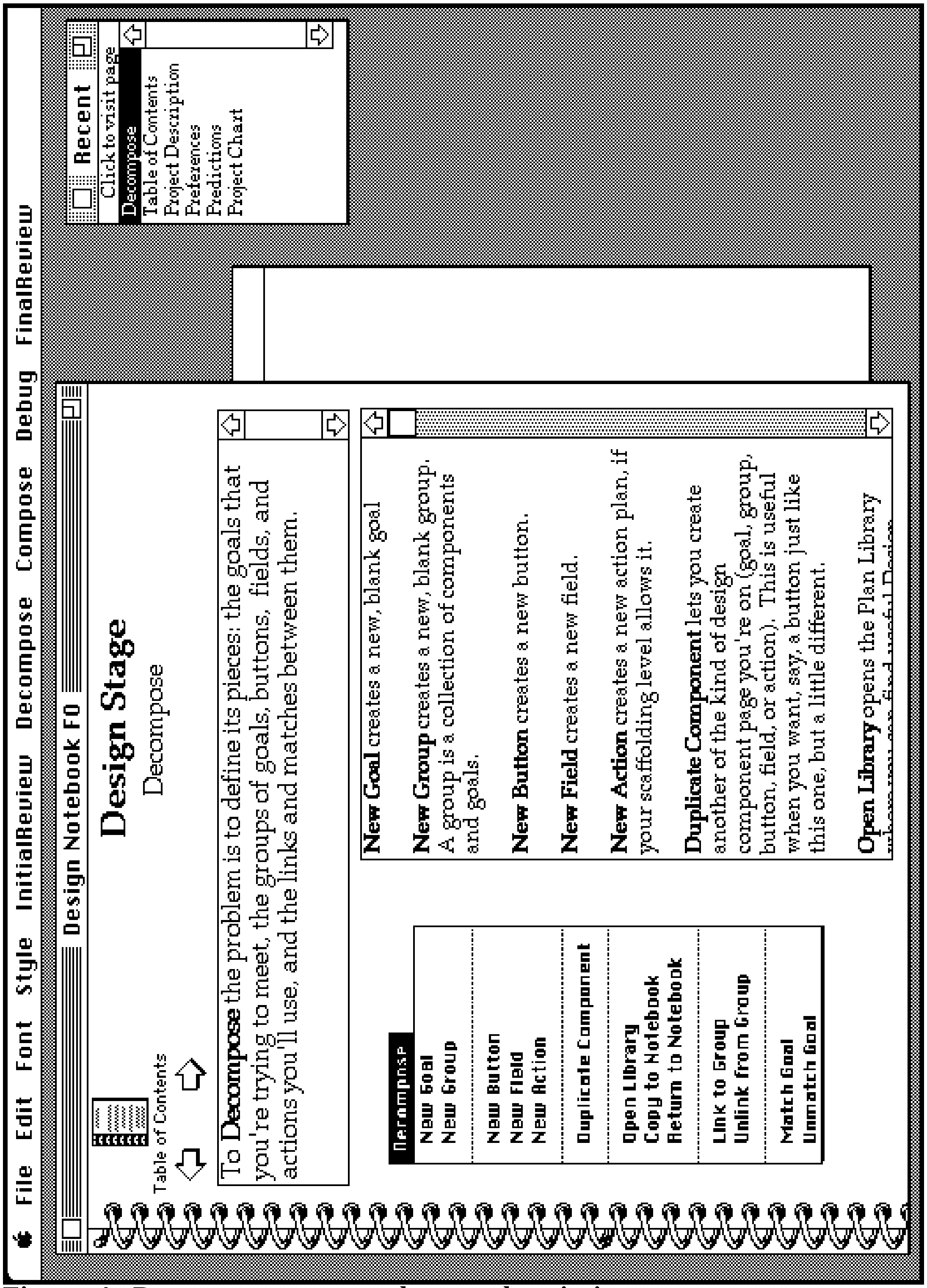

Figure 4: Decompose menu and stage description

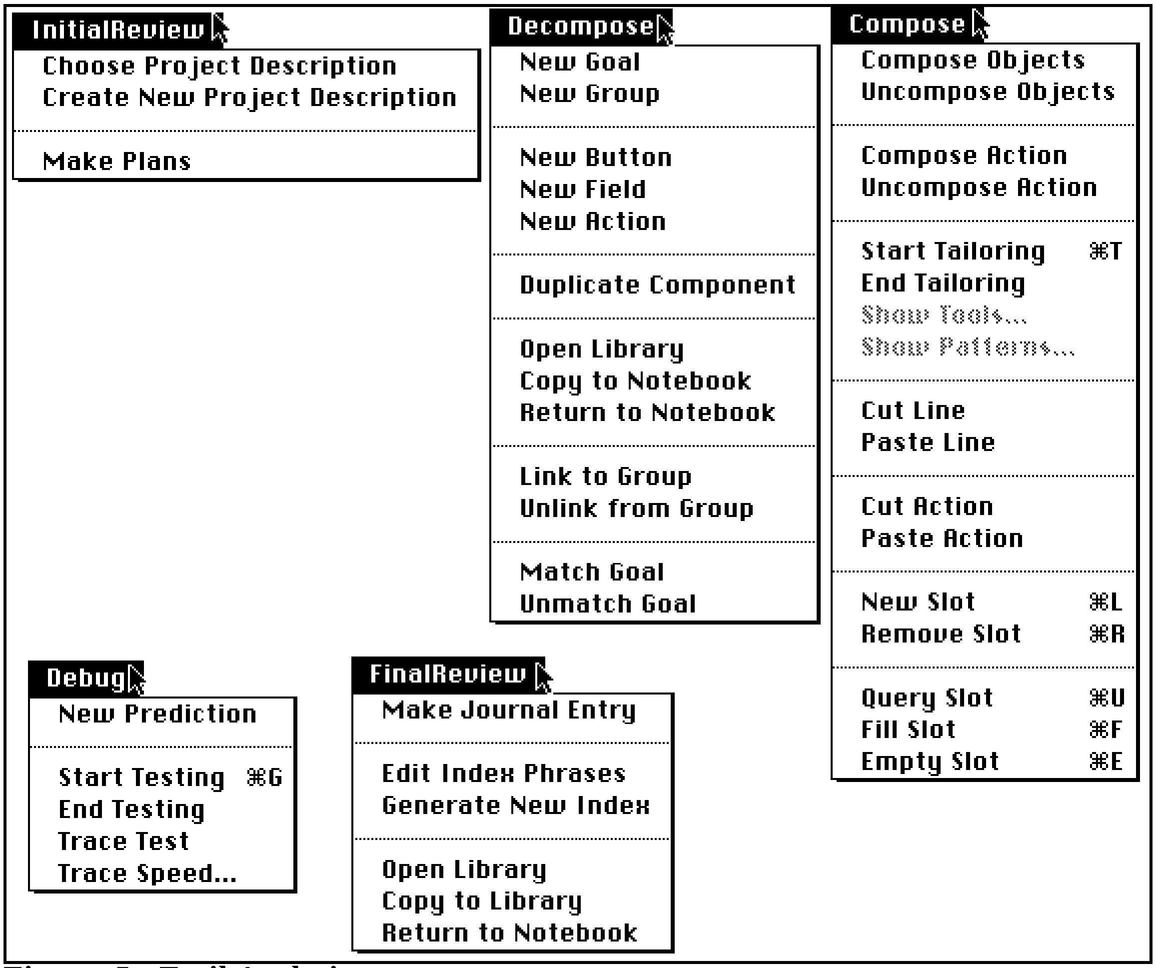

Figure 5: Emile's design stage menus

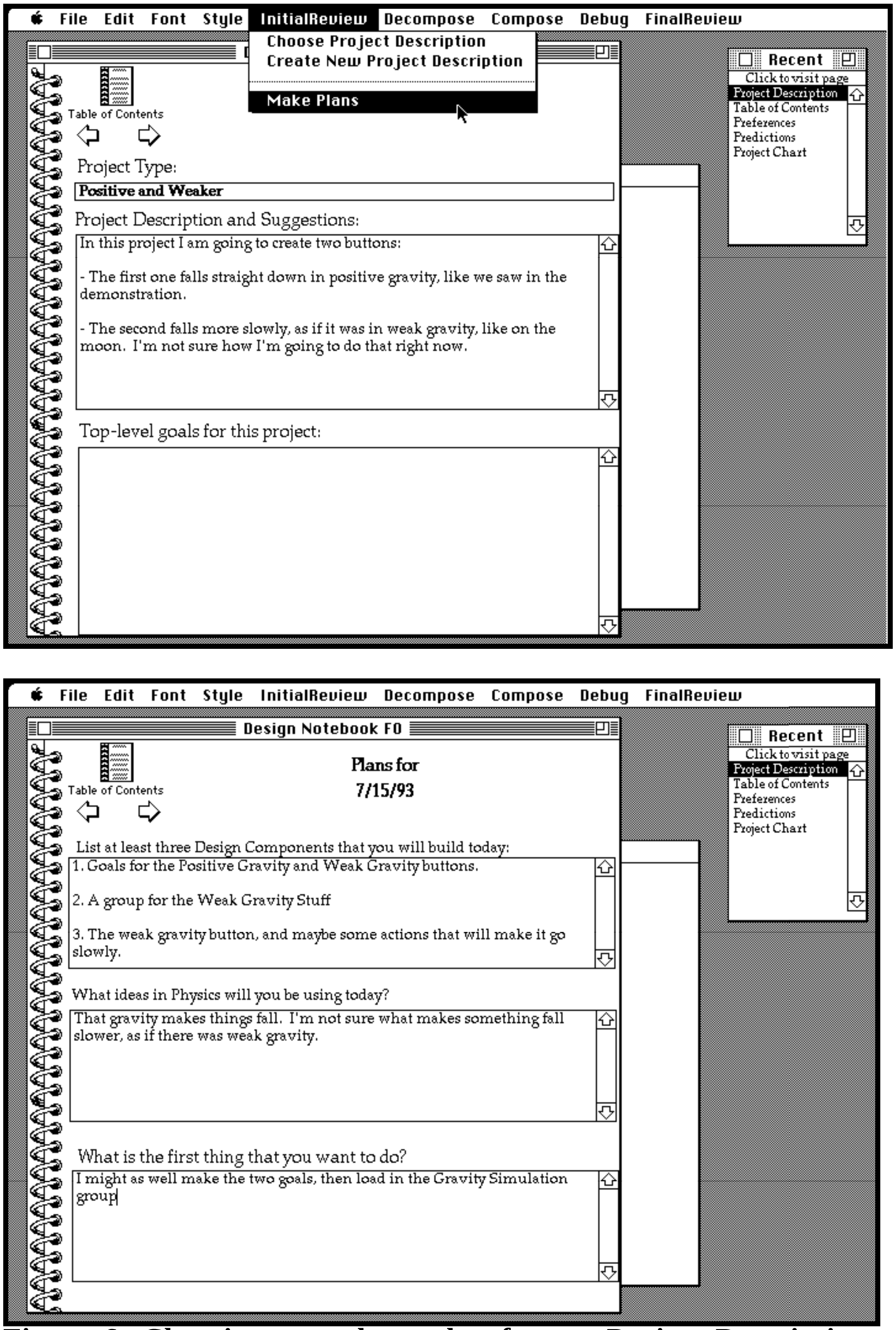

Figure 6: Choosing to make a plan from a Project Description page (a) and filling in the plan (b).

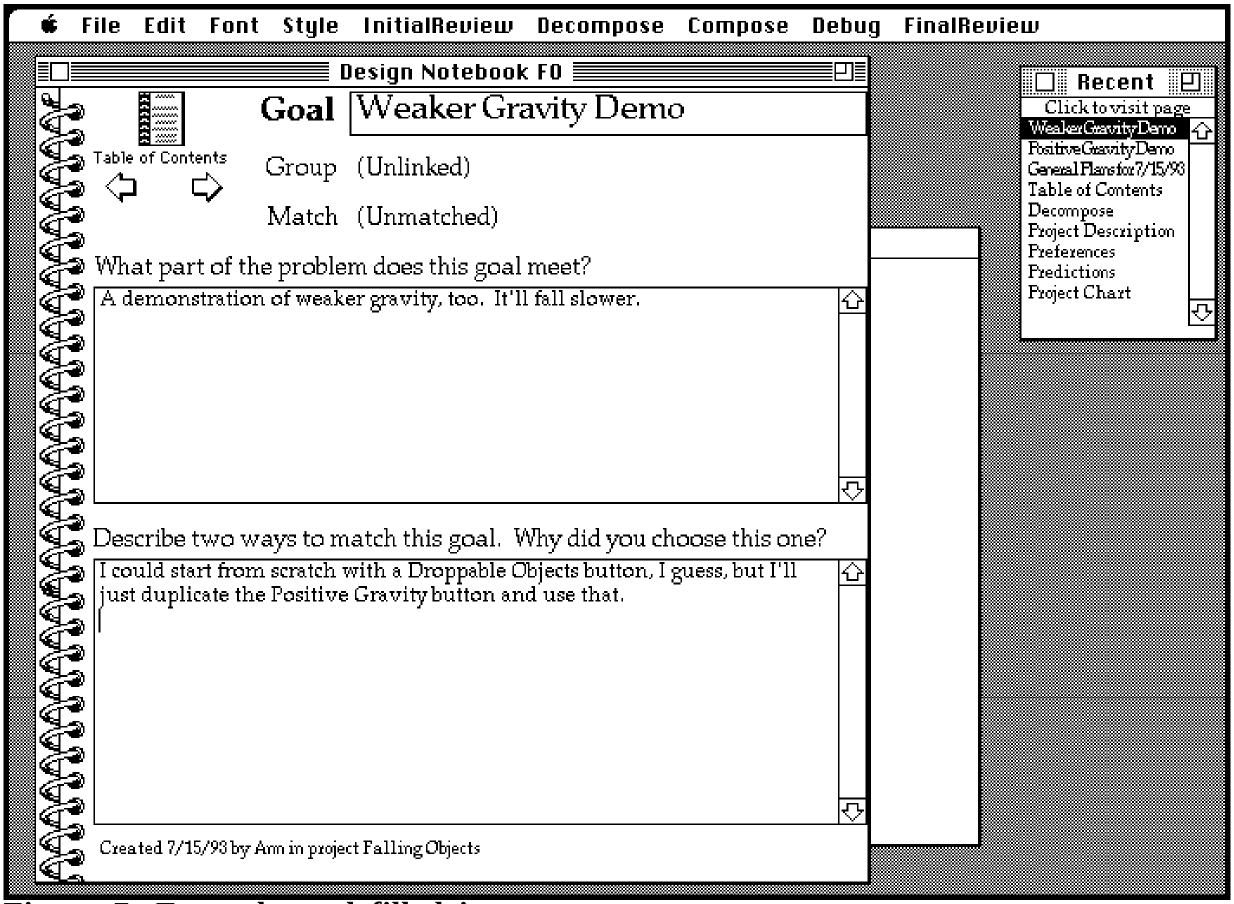

Figure 7: Example goal filled in

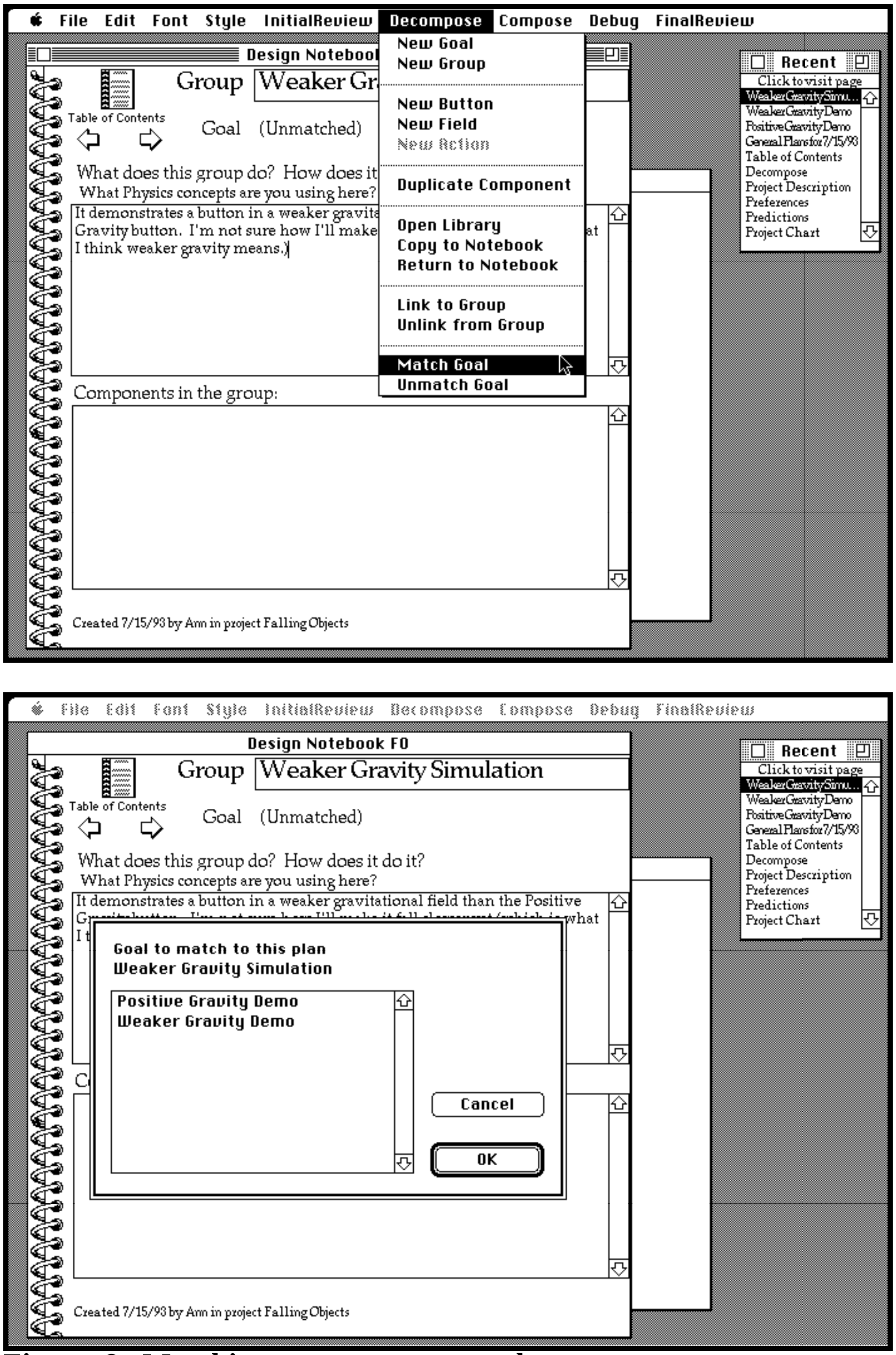

Figure 8: Matching a group to a goal

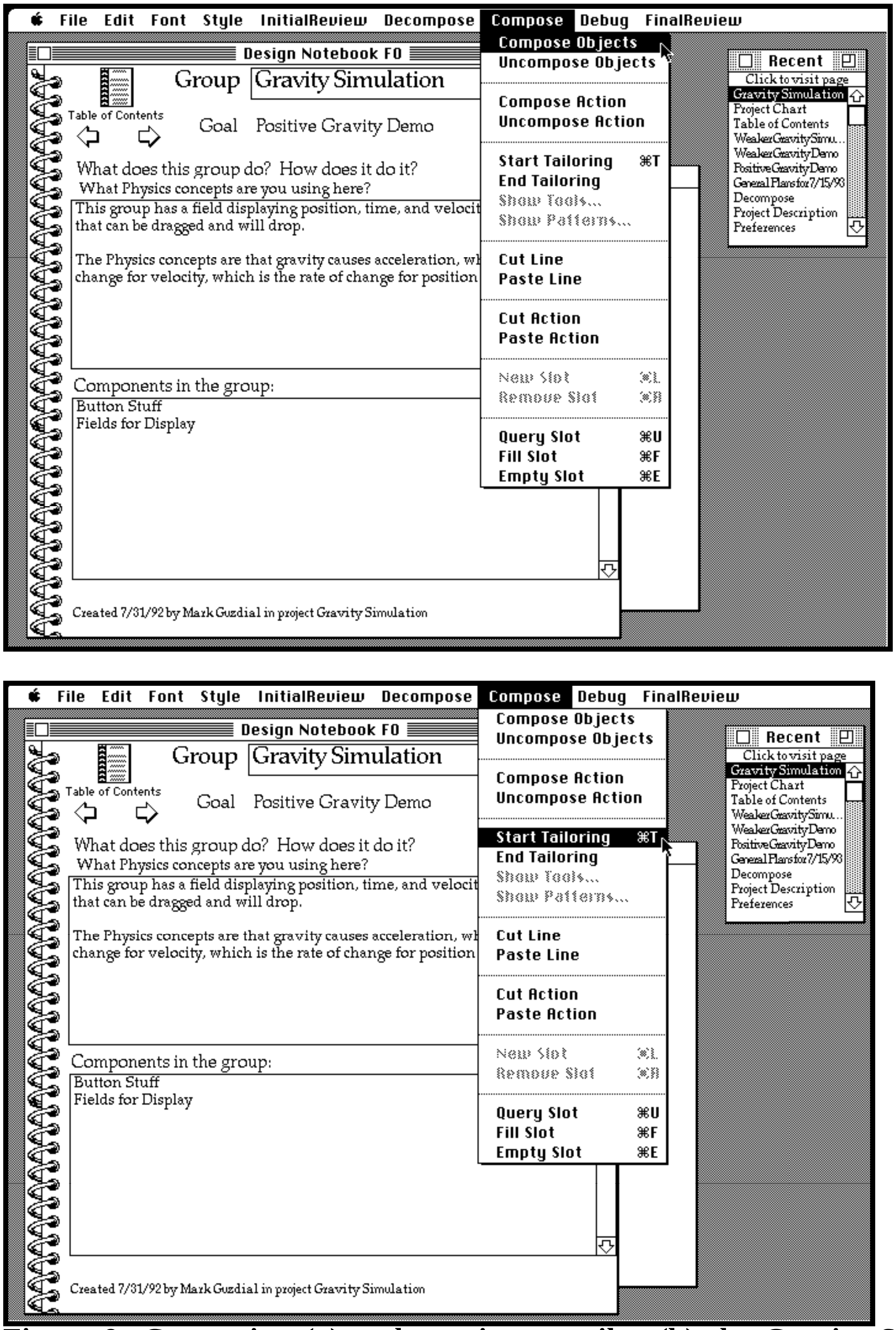

Figure 9: Composing (a) and starting to tailor (b) the Gravity Simulation group

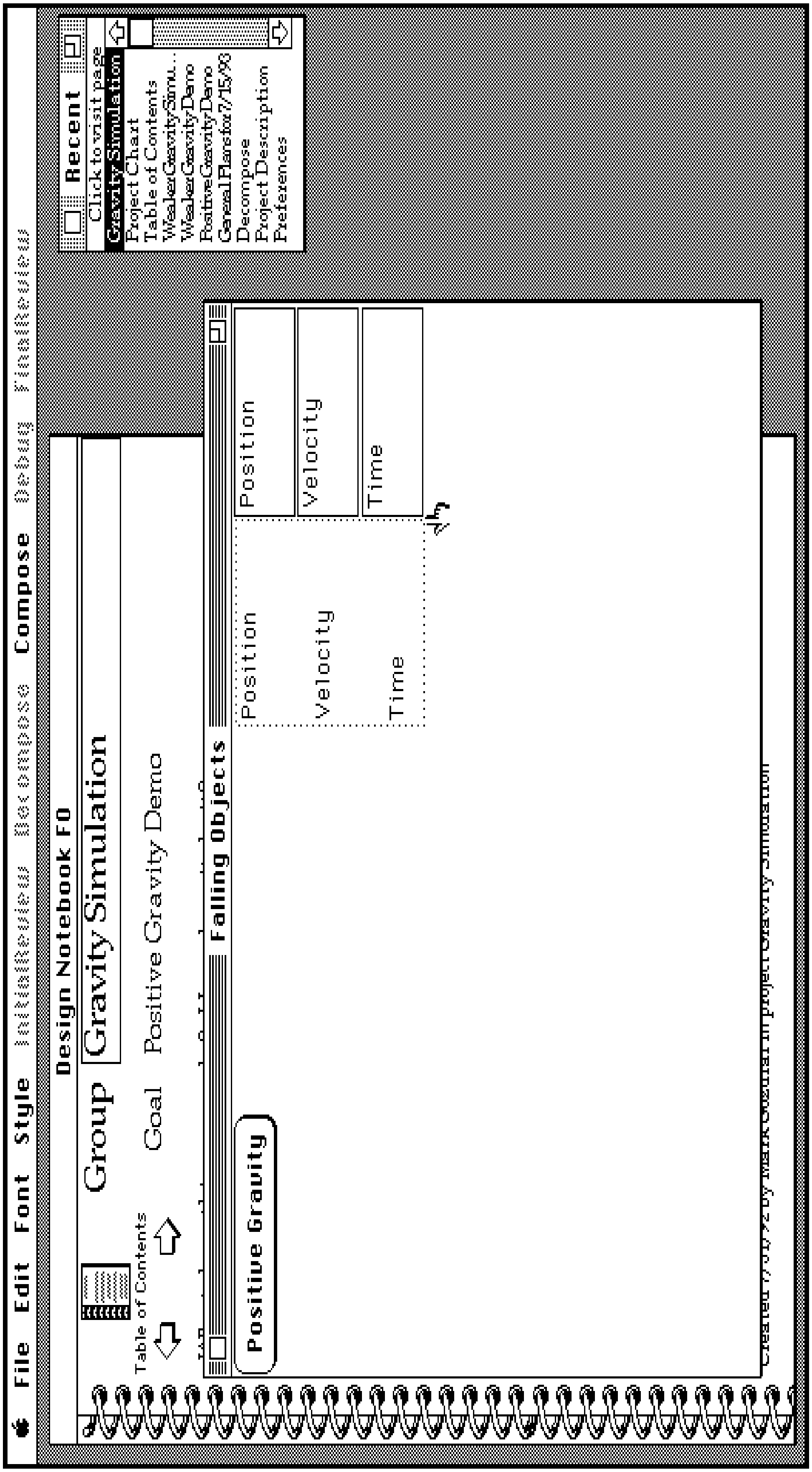
File Edit Font Style InitialReview Decompose Compose Debug FinalReview
Design Notebook F0
Table of Contents
Group
Gravity Simulation
Goal
Positive Gravity Demo
Falling Objects
Positive Gravity
Position
Velocity
Time
Position
Velocity
Time
Recent
Click to visit page
Gravity Simulation
Project Chart
Table of Contents
Decompose
Project Description
Preferences

Figure 10: Tailoring the Gravity Simulation components on the Project Window

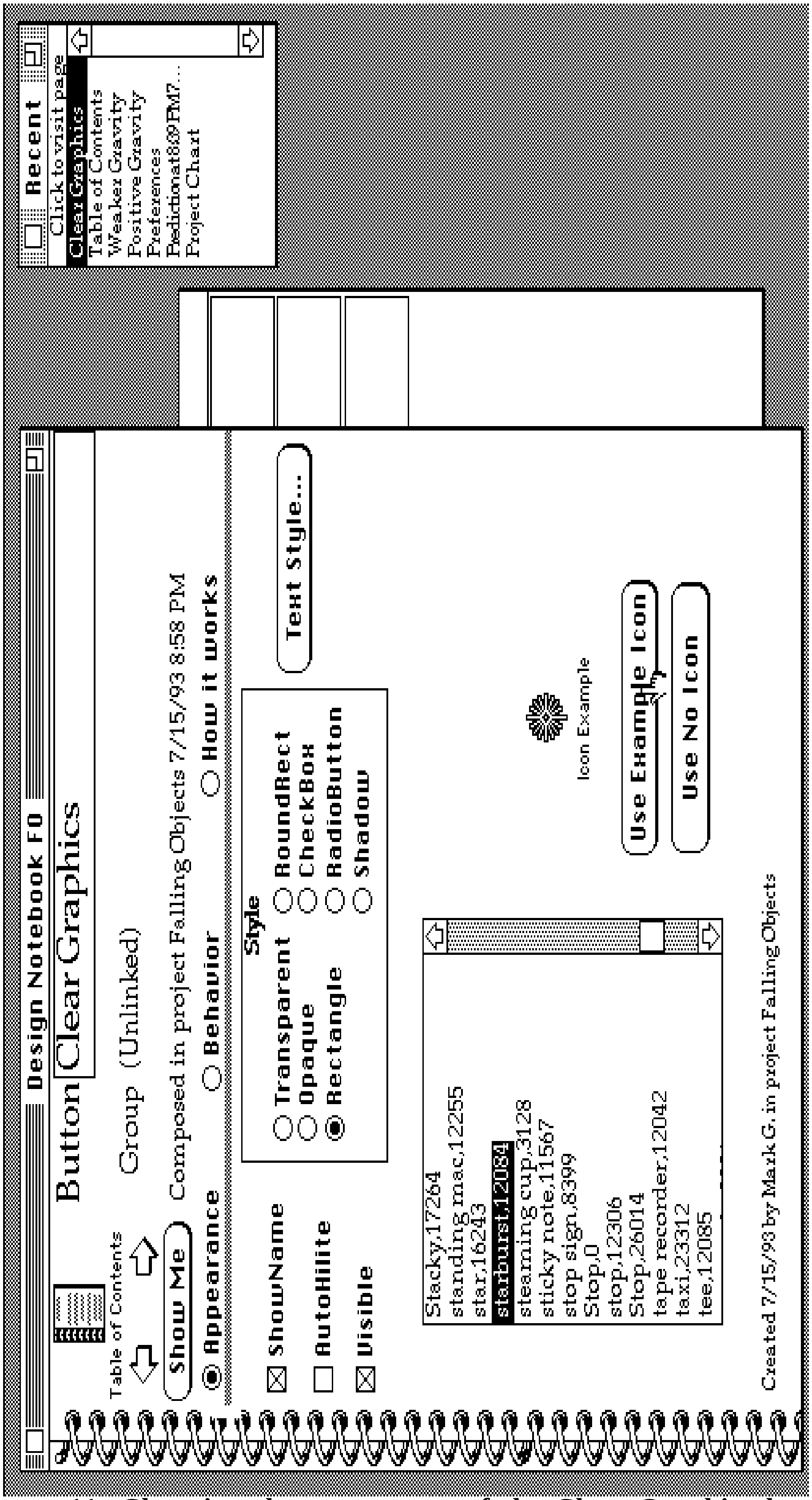

Figure 11: Changing the appearance of the Clear Graphics button

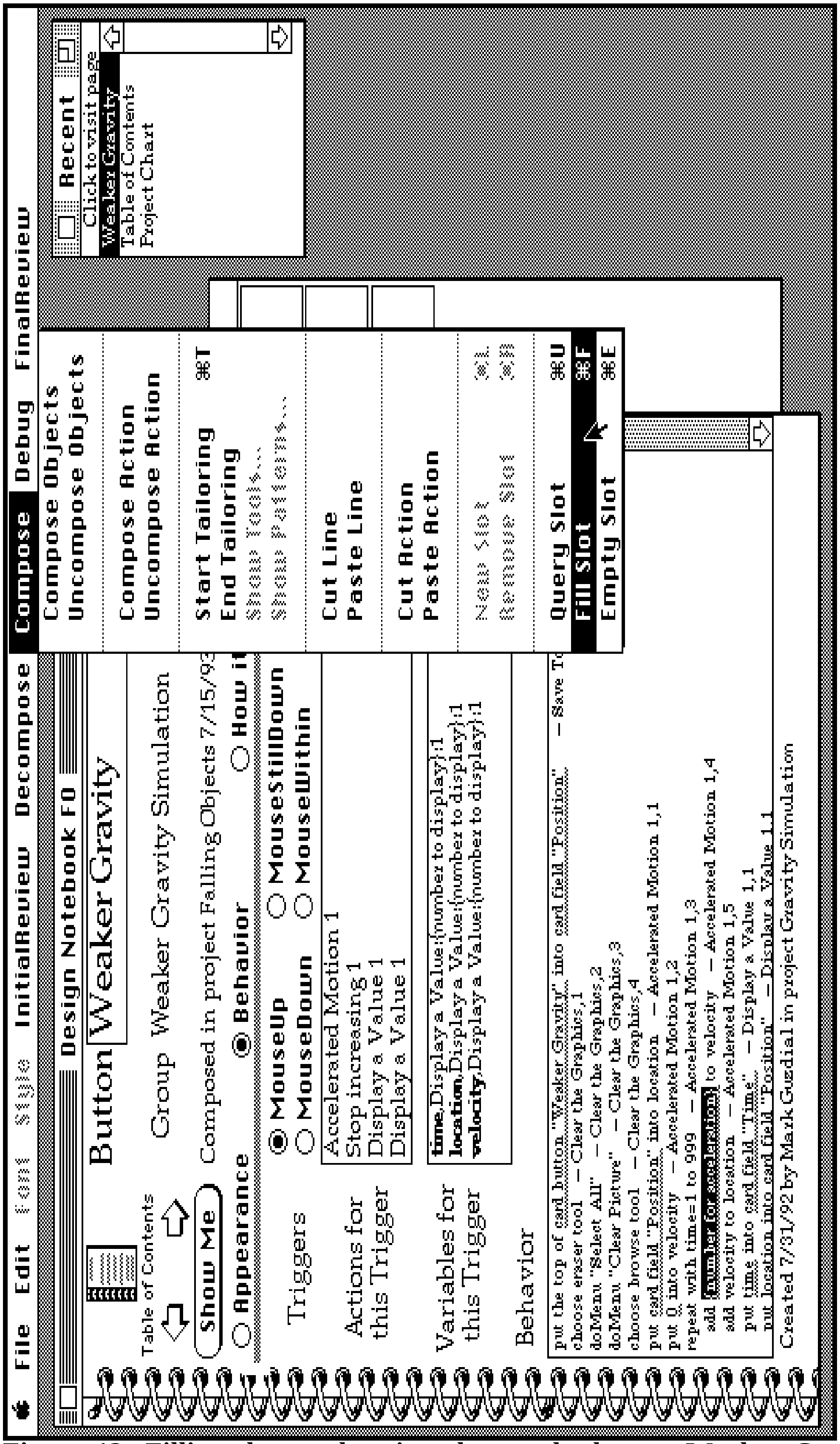

Figure 12: Filling the acceleration slot on the button Weaker Gravity

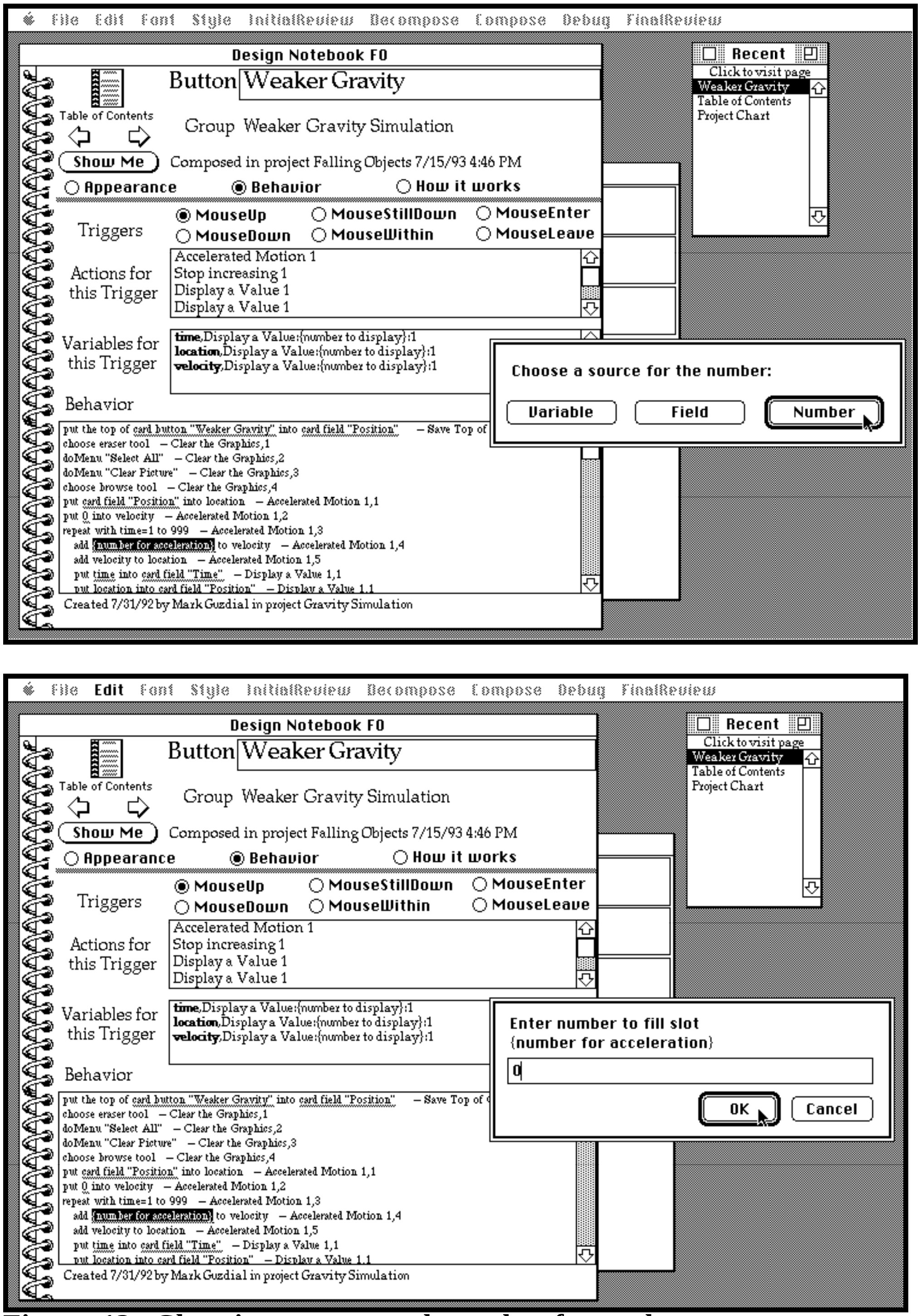

Figure 13: Choosing a type and a value for a slot

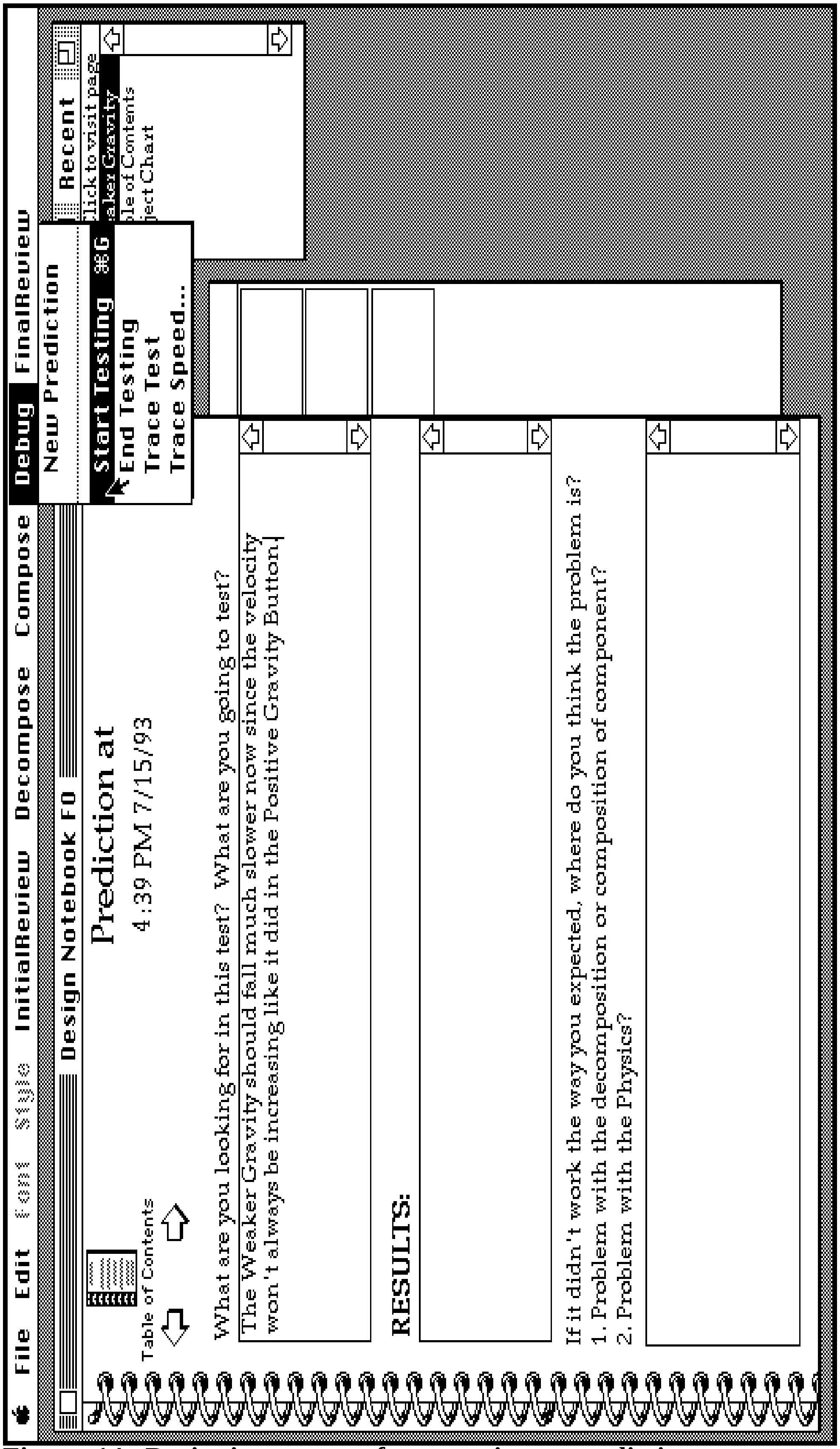

Figure 14: Beginning a test after creating a prediction

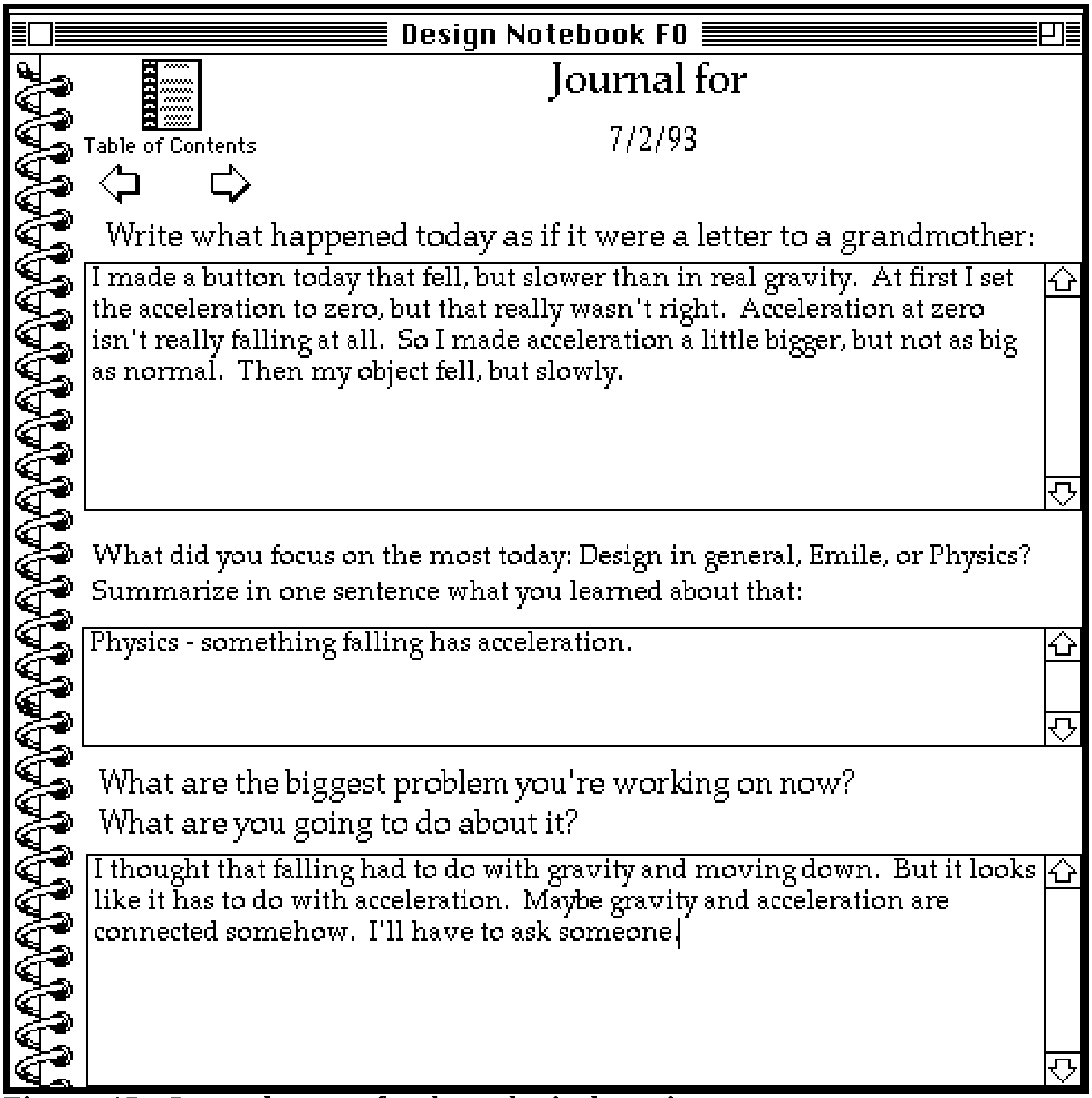

Figure 15: Journal entry for hypothetical session

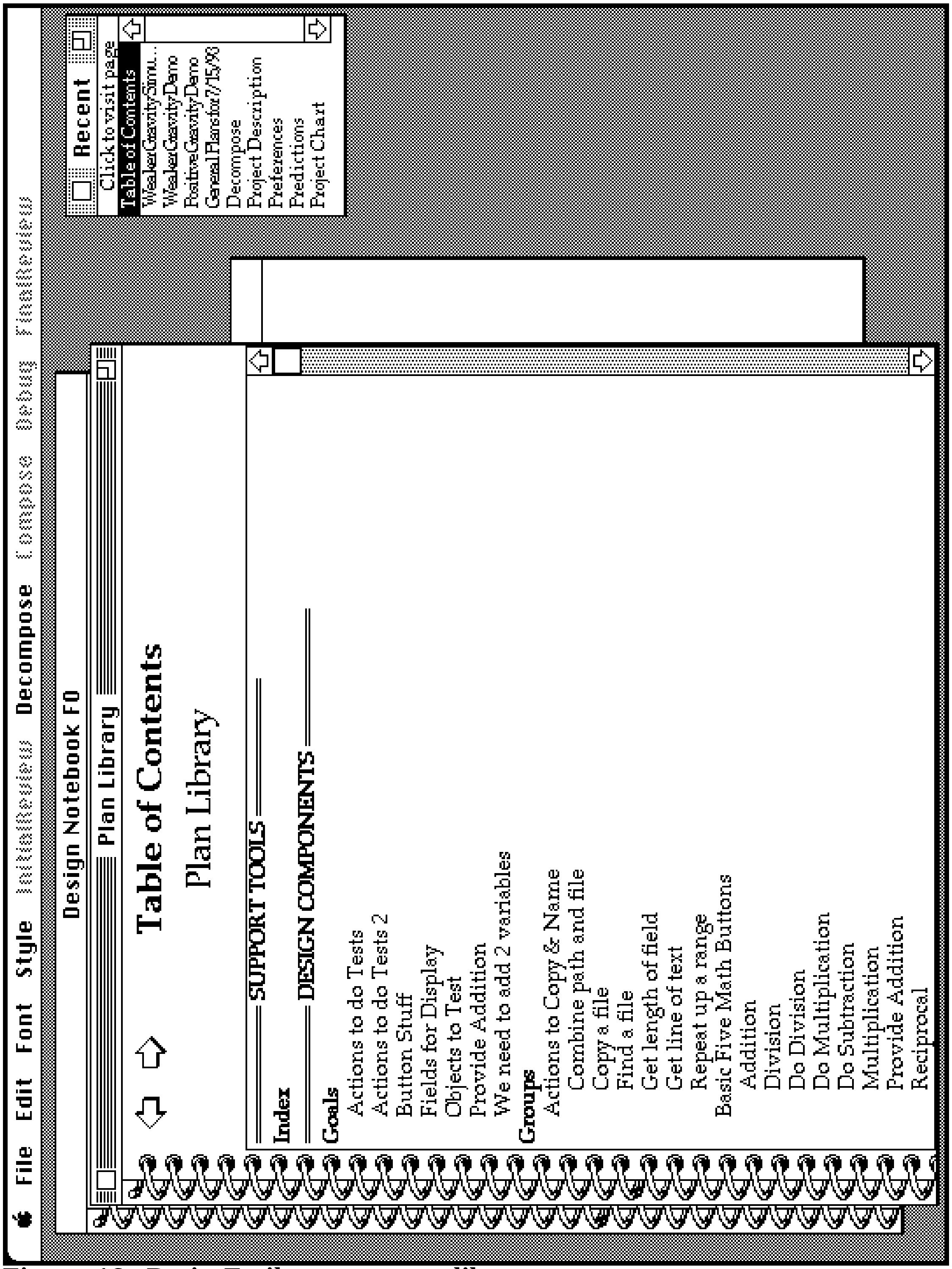

Figure 16: Basic Emile component library

Goals

- Actions to do Tests
- Actions to do Tests 2
- Button Stuff
- Fields for Display
- Objects to Test
- Provide Addition
- We need to add 2 variables

Groups

- Actions to Copy & Name
- Basic Five Math Buttons
- demonstrating drawing ovals
- Do Addition w/asking
- Fields to test
- File Actions
- File Actions 2
- Gravity Simulation
- Image Buttons
- Multimedia Buttons
- Pick from noun and verb fields
- Pos/V/T/Labels
- Positive Acceleration
- SCSI Actions
- Three multi-line tests
- Three Test Buttons
- Three Test Buttons 2
- Tools for Sampling Keyboard
- Tools for Simulating Velocity
- Trignometric Functions
- visual effects group

Fields

- A Noun Phrase
- A Verb Phrase
- Current Visual Effect
- File List
- First
- Labels
- Noun Phrases
- Position
- Second
- Sentence
- Time
- Velocity
- Verb Phrases

Buttons

- A Droppable Object
- Addition
- CD Stop
- Clear Screen
- Division
- Do Demo
- Draw Ovals
- File Button
- First < Second
- First < Second 2
- First = Second
- First > Second
- Generate Random Sentence
- GraphicImage
- Multiplication
- Picture Show
- Play one Track
- Play Scale
- Play Sound
- Play Video
- Positive Gravity
- QuickTime On-Screen
- QuickTime Window
- Reciprocal
- Record Sound
- Rocket
- Stop Video
- Subtraction
- VideoSequence
- Visual Effects

Actions

- Accelerated Motion
- Accelerated Motion 1
- Add to Field
- Ask the user for Something
- Ask the user for Track
- Clear the field
- Clear the Graphics
- Clear the Whole Screen
- Clear the Whole Screen 2
- Combine path and file
- Combine two pieces of text
- Convert X to HC
- Convert Y to HC
- Copy a file
- Display a Value
- Display a Value 1
- Dissolve effect
- Do a bunch of ovals
- Do Addition
- Do Division
- Do Multiplication
- Do Subtraction
- Drag-and-Drop
- Drag-and-Drop 1
- Draw a Line
- Draw a line with 2 points
- Draw a rectangle
- Draw an Oval
- Draw X Axis
- Draw Y Axis
- Find a file
- Find SCSI Info
- Get a random number
- Get contents of folder
- Get contents of folder 2
- Get e^power
- Get File Creation Date
- Get File Type
- Get length of field
- Get line of text
- Get name of SCSI drive
- Get path to folder
- Get SCSI Number for drive
- Get the arctangent
- Get the cosine
- Get the first number
- Get the natural log
- Get the Power
- Get the second number
- Get the sine
- Get the square root
- Grab a random Noun Phrase
- Grab a random Verb Phrase
- Hide an object
- Increment Value
- Increment Value 1
- Iris close effect
- Iris Open Effect
- Iris Open Effect 2
- Leave the Repeat
- List SyQuest Drives
- Play a CD track
- Play a QT in Window
- Play a Sound
- Play a videodisc segment
- Play a videodisc segment 2
- Play QT on-screen
- Play some notes
- Put words into field
- Record a Sound from Mic
- Repeat down a range
- Repeat up a range
- Repeat up a range 2
- Save the left of button
- Save the left of button 1
- Save the location
- Save Top of Object
- Save Top of Object 1
- Save Top of Object 2
- Say Something
- Set the bottom
- Set the left of button
- Set the left of button 1
- Set top of object
- Set top of object 1
- Set top of object 2
- Show a graphic
- Show a movie
- Show a Picture
- Show an object
- Shrink to center effect
- Sound a beep
- Stop increasing
- Stop increasing 1
- Stop the CD
- Stop Videodisc
- Store a number to a field
- Tell the user something
- Tell the user something 2
- Tell the user something 3
- Test for equality
- Test for greater-than
- Test for greater-than 2
- Test for less-than
- Test for less-than 2
- Trace with dots
- Truncate to integer
- Wait a little
- Wipe left effect

Table 3: Table of contents in basic Emile library

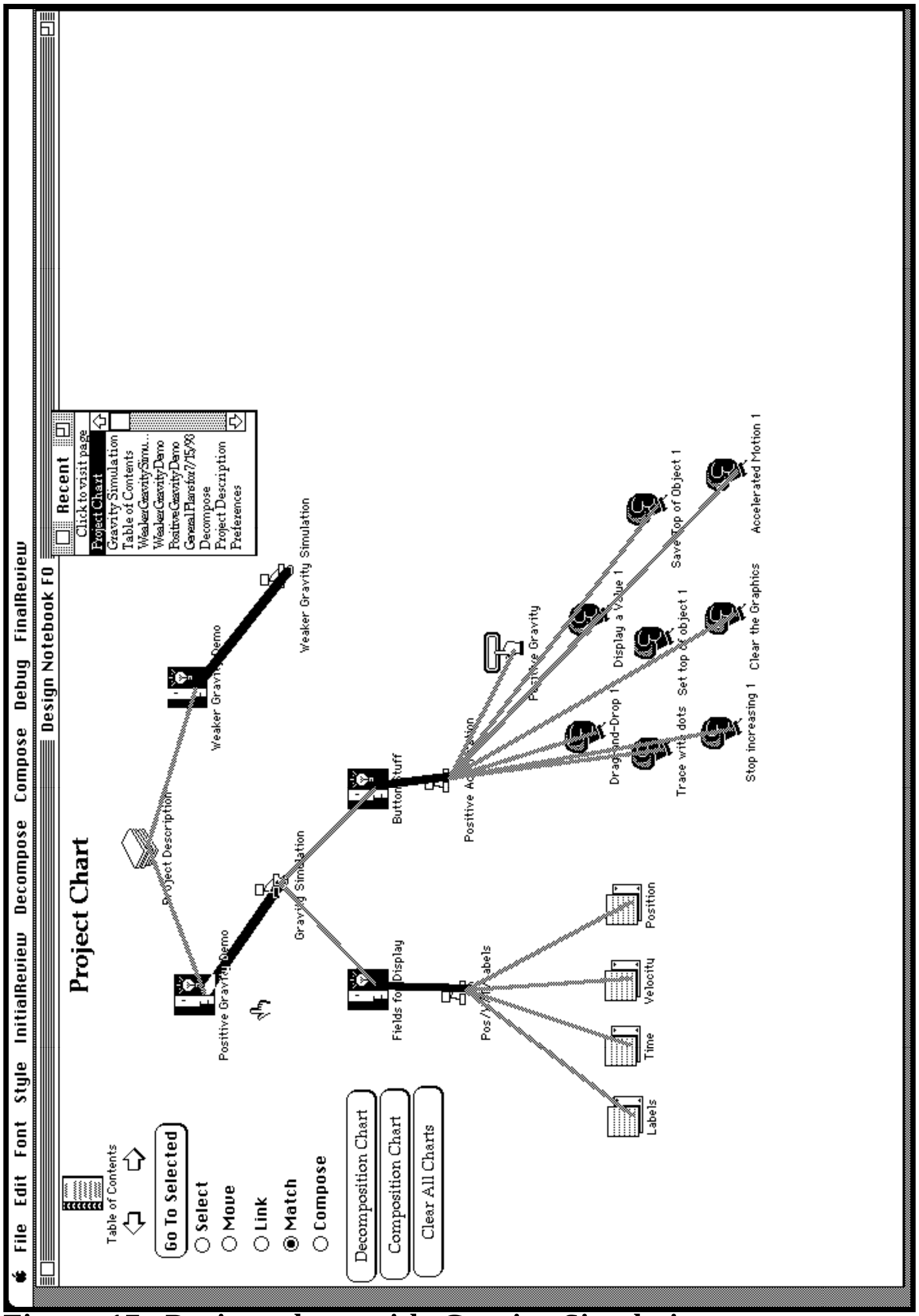

Figure 17: Project chart with Gravity Simulation group

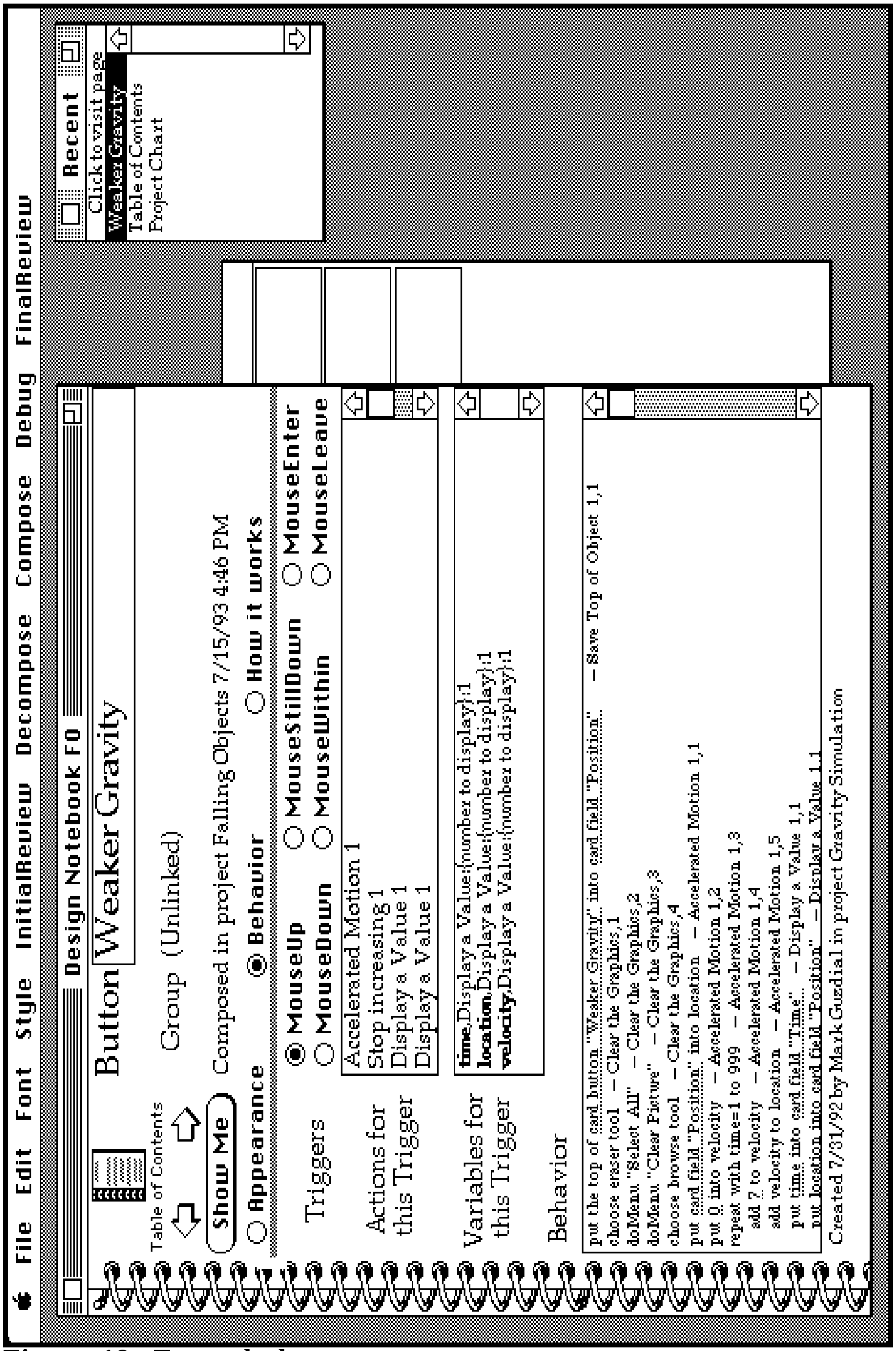

Figure 18: Example button page

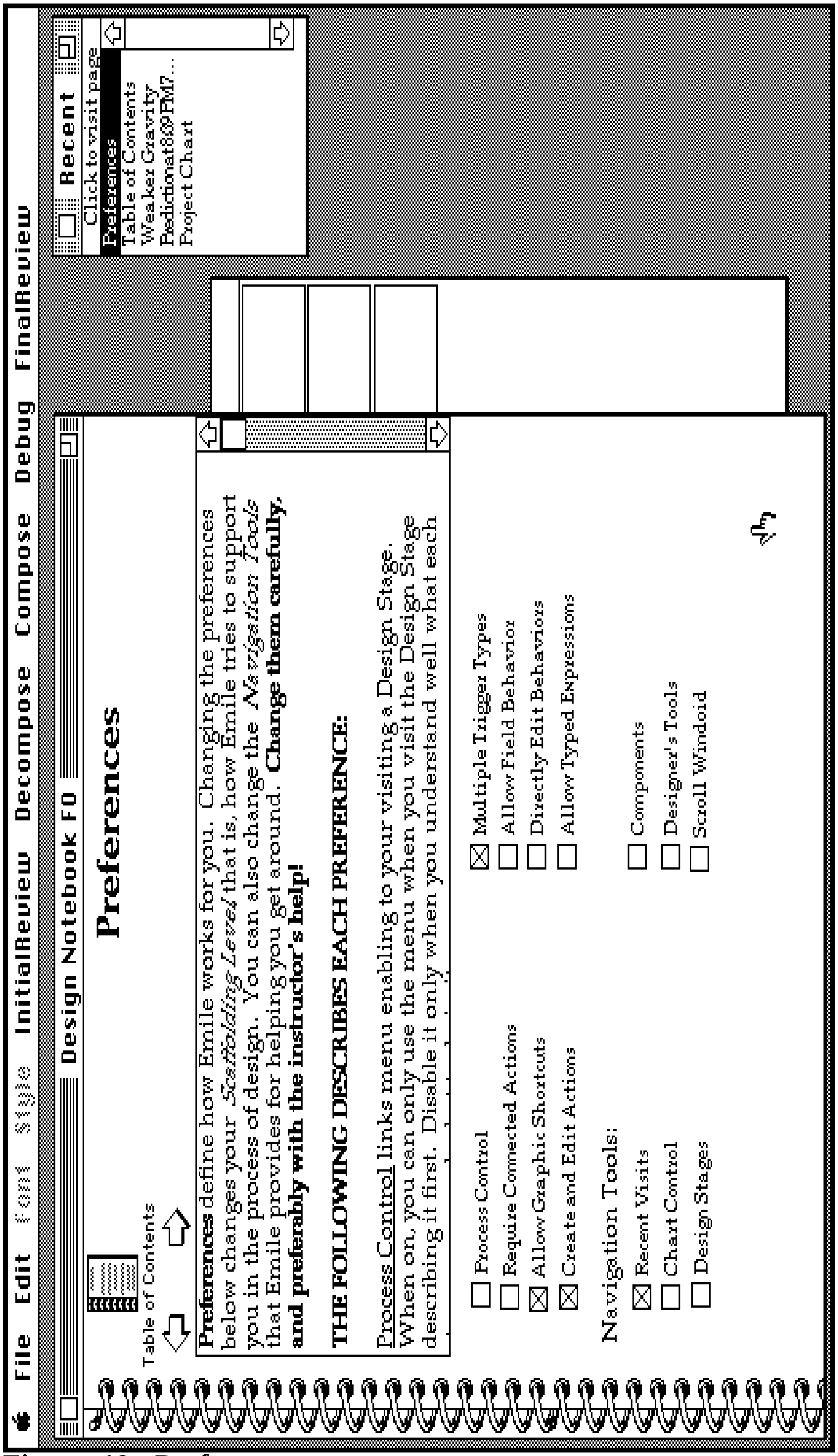

Figure 19: Preferences page

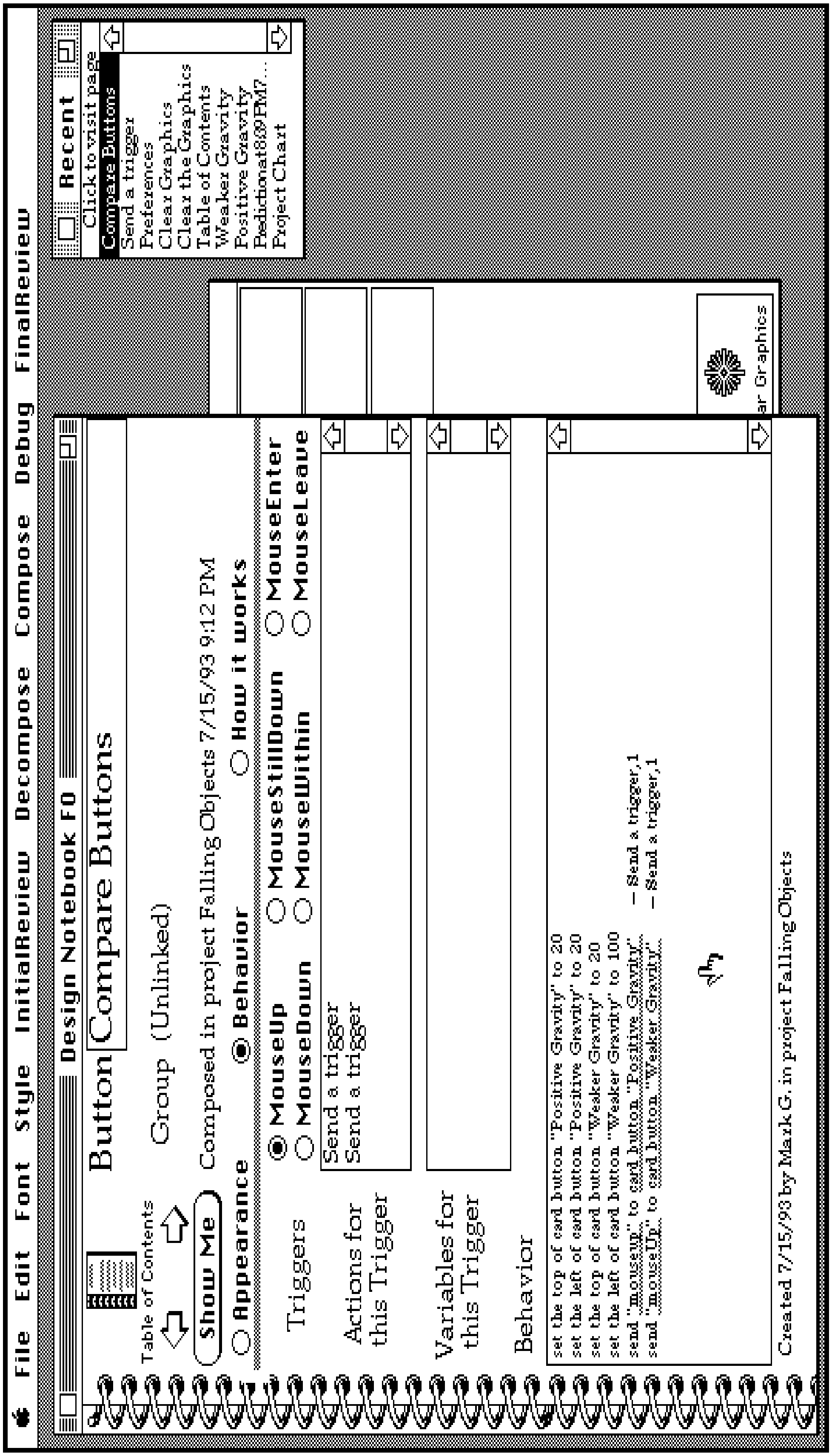
File Edit Font Style InitialReview Decompose Compose Debug FinalReview
Design Notebook F0
Button Compare Buttons
Table of Contents
Group (Unlinked)
Show Me
Composed in project Falling Objects 7/15/93 9:12 PM
Appearance
Behavior
How it works
Triggers
MouseUp
MouseDown
MouseStillDown
MouseWithin
MouseEnter
MouseLeave
Actions for this Trigger
Send a trigger
Send a trigger
Variables for this Trigger
Behavior
set the top of card button "Positive Gravity" to 20
set the left of card button "Positive Gravity" to 20
set the top of card button "Weaker Gravity" to 20
set the left of card button "Weaker Gravity" to 100
send "mouseup" to card button "Positive Gravity" -- Send a trigger,1
send "mouseUp" to card button "Weaker Gravity" -- Send a trigger,1
Created 7/15/93 by Mark G. in project Falling Objects
Recent
Click to visit page
Compare Buttons
Send a trigger
Preferences
Clear Graphics
Clear the Graphics
Table of Contents
Weaker Gravity
Positive Gravity
Prediction at 8:39 PM7...
Project Chart
ar Graphics

Figure 20: Combining library actions with the typed statements

Scaffolding Category	**Emile Scaffolding Feature**	**How Fades**
Communicating Process	*Macro*	
	• Design stages	Does not fade
	• Design Stage Pages	Does not fade
	• Menu names	Does not fade
	Micro	
	• Actions and Slots	Levels
	• Goals and Groups	Voluntary less use
	• Design Notebook	Levels
	• Menu items	Does not fade
	• Library	Voluntary less use
	• Representations	Levels
Coaching	*Macro*	
	• Stage Prompting	Immediate stop
	Micro	
	• Top-Down Design Enforcement	Immediate Stop
Eliciting Articulation	*Macro*	
	• Project Descriptions	Voluntary less use
	• Plans	Voluntary less use
	• Goals	Voluntary less use
	• Predictions	Voluntary less use
	• Journals	Voluntary less use
	Micro	
	• Naming	Does not fade

Table 4: Fading in Emile's Software-Realized Scaffolding

Students	**Age**	**Grade in Fall**	**Program-ming Experience**	**Physics Experience**
B	14	9	None significant	Jr. High Physical Science
C	18	<Graduated>	None	None
L	16	11	None significant	None
M	15	10	Knew Basic, Pascal, and C	None
S	14	9	Knew Basic and C	Jr. High Physical Science

Table 5: Summary of student prior experiences in programming and physics

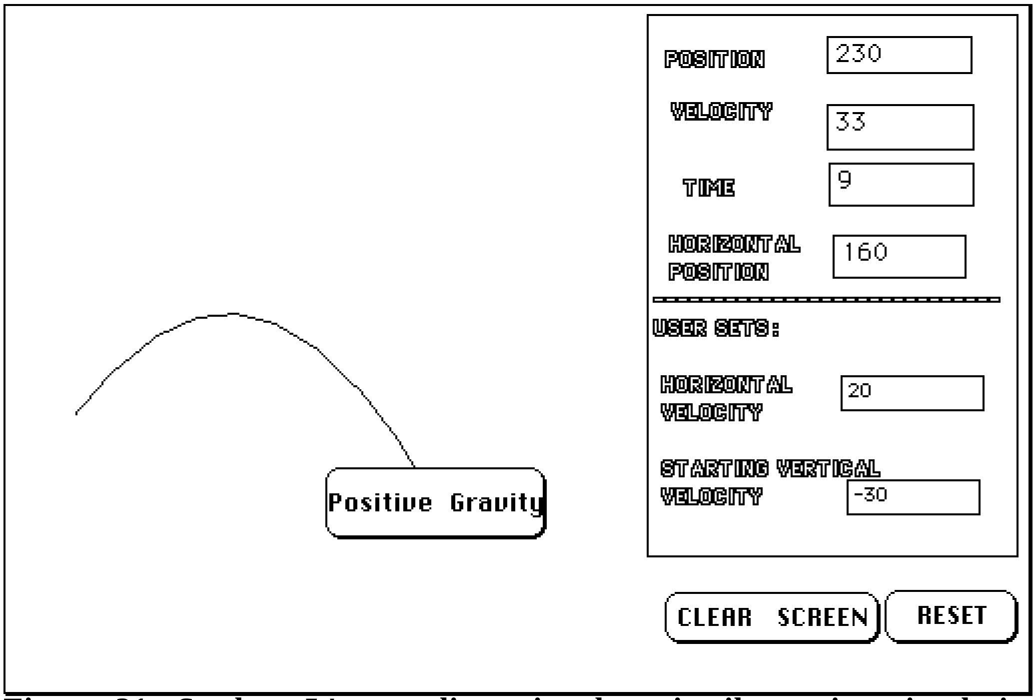

Figure 21: Student L's two-dimensional projectile motion simulation

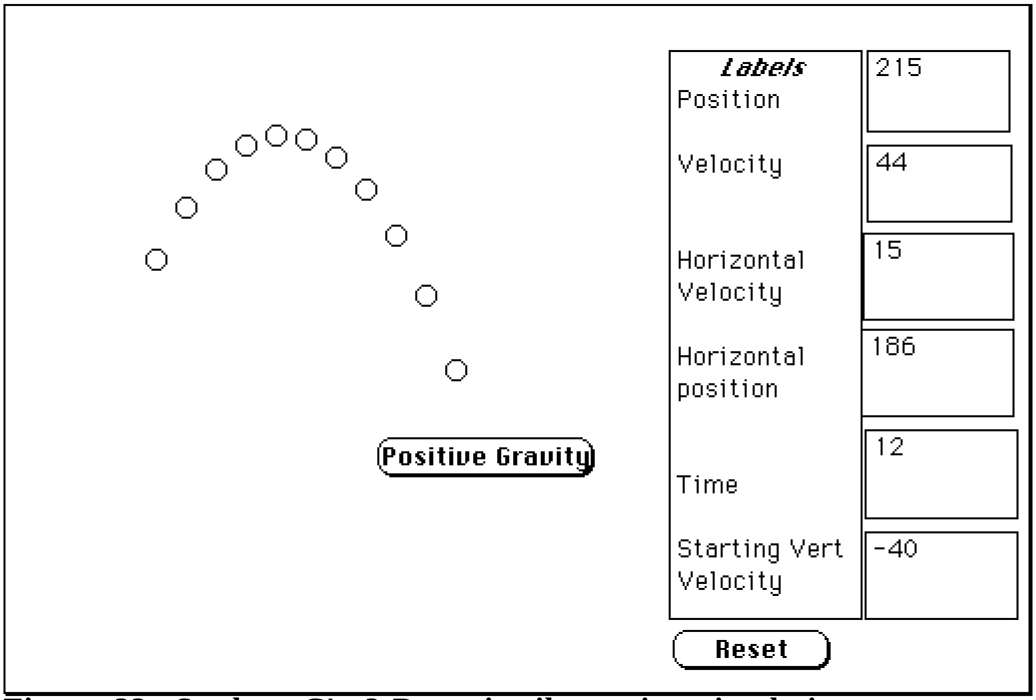

Figure 22: Student C's 2-D projectile motion simulation

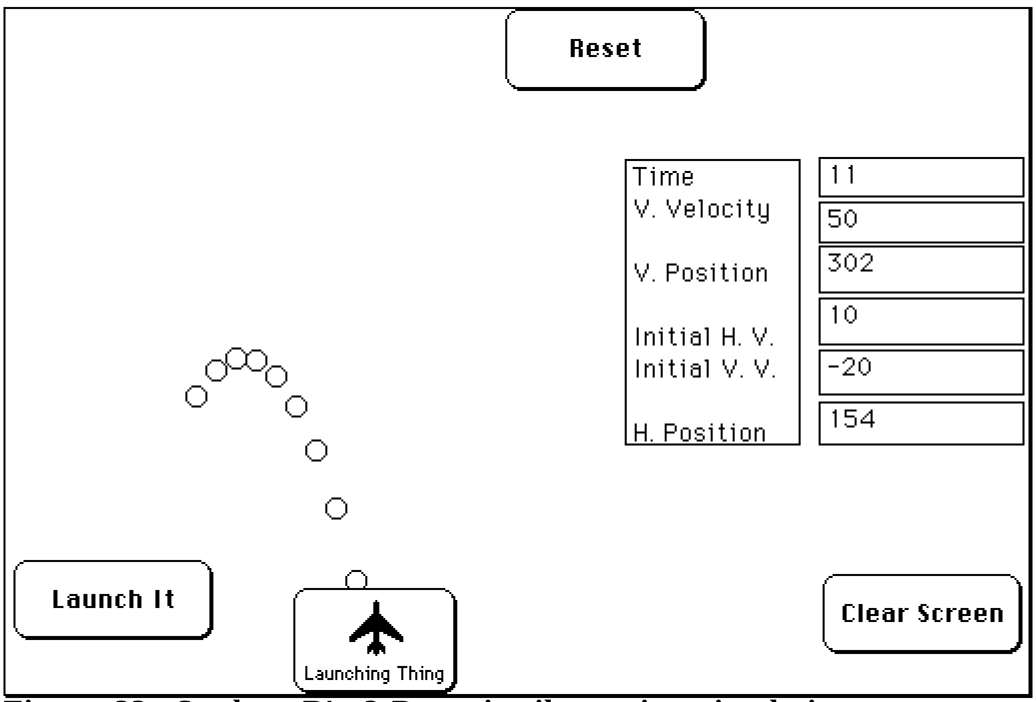

Figure 23: Student B's 2-D projectile motion simulation

	B	C	L	M	S
Project 1					• Type expressions for slots • Create actions • Directly edit behaviors
Project 2				• Type expressions for slots • Create actions • Directly edit behaviors	
Project 3					
Project 4	• Directly edit behaviors		• Create actions		

Table 6: Projects at which students changed scaffolding controlling behavior construction

	Project 1		**Project 2**		**Project 3**		**Project 4**	
	Prog	M-B	Prog	Physics	Prog	M-B	Prog	M-B
B	√	√	√	√	√	√	√	√
C	√	–	√	√	√	√	√	√
L	√	√	√	–	√	√	√	√
M	–	√	√	–	–	√	√	√
S	√	√	√	√	–	–	√	√

Table 7: Student performance on projects
(√ = successful performance, – = partially or not successful)

	B	C	L	M	S
Concept of velocity	2->3	1->2	1->1	->3	3->3
Concept of acceleration	1->2	1->2	1->2	->2	2->3
Concept of projectile motion	2->3	1->2	1->2	->3	2->2

Table 8: Summarizing physics learning (Pre-level -> Post-level)