Learning Relational Algebra by Snapping Blocks

Jason Gorman
Dept of Computer Science
James Madison University
Harrisonburg, Virginia, USA
gorma2jp@dukes.jmu.edu

Sebastian Gsell
Dept of Computer Science
James Madison University
Harrisonburg, Virginia, USA
gsellsl@dukes.jmu.edu

Chris Mayfield
Dept of Computer Science
James Madison University
Harrisonburg, Virginia, USA
mayfiecs@jmu.edu

ABSTRACT

Relational algebra provides a theoretical foundation for how modern database management systems optimize and execute queries. Its main concepts are based on set theory and first order logic, which can be challenging for students to learn due to their abstract nature. This paper presents Bags, a new type of visual programming environment (inspired by Snap!) for the teaching of relational operations and data analysis. Students formulate algebraic queries by snapping together graphical blocks that represent data sets and relational operators, resulting in an interactive visualization of the underlying concepts. The outcomes of this work will not only enhance university-level database courses, but also provide an engaging computational thinking resource for K-12 teachers in content areas outside of science and engineering.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer Science Education

General Terms

Design, Theory

Keywords

Computational thinking, data analysis, query design

1. INTRODUCTION

Over the past decade, Computer Science education researchers have developed drag-and-drop environments for teaching computer programming [8]. Notable examples include Alice (CMU), Scratch (MIT), Greenfoot (Kent), and App Inventor (Google). Although each project has its own emphasis (e.g., object oriented design, interaction with media, simulation and games, mobile development), they all focus on teaching introductory programming concepts. Students learn about instructions, decisions, loops, variables, operators, and so forth by manipulating "blocks" that represent these structures. This approach has a number of pedagogical advantages, including simplifying the programming environment, encouraging self-directed learning, making the execution of programs visual, eliminating syntax error messages, and making data more concrete [5].

In this paper we present Bags,[1] a new type of drag-and-drop environment suitable for teaching relational algebra. In contrast to current tools like Scratch and Alice that allow students to interact with computation, our software will enable students to explore and manipulate data. Students formulate database queries by combining blocks representing relational operations, including selection, projection, inner/outer join, grouping, aggregation, and sorting. As in [10], we are developing an online introduction to fundamental database theory concepts. But our approach allows users to evaluate ad hoc queries in real-time, giving them an interactive visualization of the data as they add or remove individual operations. Students are also able to experiment with different configurations of blocks and observe the corresponding query results.

What we are proposing is very different from graphical query editors, such as the visual designer in Microsoft Access. Students are not writing declarative queries that have a straightforward translation to SQL. In contrast, they are building relational algebra expressions based on set theory and first order logic, which in turn are evaluated sequentially. As in [2], the goal is for students to understand how database management systems apply selections, projections, and joins to answer queries over relational data. Query optimizers use similar algebraic representations to analyze and evaluate query execution plans. In a sense, students play the role of the optimizer while exploring data of their choice.

Bags will not only benefit database courses at the university level, but also have significant impact on K-12 education [1]. Recent efforts have developed new curricula for middle school and high school students to explore computational thinking (see e.g., [6]). Most activities for teaching computational thinking—available in the literature and on the Internet—have been successfully integrated in after-school programs with a science or technology emphasis [9]. This project aims to provide an engaging computational thinking resource for K-12 teachers in other content areas, such as social studies or humanities. For example, students in a geography course may use Bags to explore census or world

SIGCSE'14, March 5–8, 2014, Atlanta, GA, USA.

ACM 978-1-4503-2605-6/14/03 ...$15.00.
http://dx.doi.org/10.1145/2538862.2538961.

[1]The name of our environment makes reference to set theory, i.e., a bag is a multiset. SQL and other database languages are based on bag semantics.

RANKING	MOVIE	DISTRIBUTOR	RELEASE YEAR	GROSS (USD in millions)	ADJUSTED GROSS (USD in Millions)
1	Gone with the Wind	MGM	1939	198.68	1855.13
3	The Sound of Music	Fox	1965	163.21	1224.45
6	The Ten Commandments	Paramount	1956	85.40	1037.70
8	Snow White and the Seven Dwarfs	Disney / RKO	1937	184.93	943.39
9	Doctor Zhivago	MGM	1965	111.72	807.84
11	Ben-Hur	MGM	1959	74.00	789.56
15	Mary Poppins	Disney	1964	102.27	731.31
21	The Graduate	AVCO Embassy	1967	104.90	653.29
27	My Fair Lady	Warner Bros.	1964	72.43	582.75
30	Around the World in Eighty Days	United Artists	1956	42.00	558.62
36	Butch Cassidy and the Sundance Kid	Fox	1969	102.31	530.34
37	The Robe	Fox	1953	36.00	519.71
40	Cleopatra	Fox	1963	57.78	507.90
44	101 Dalmatians	Disney	1961	144.88	490.51
46	Thunderball	United Artists	1965	63.60	486.27
53	Lady and the Tramp	Disney	1955	88.30	471.05
56	The Greatest Show on Earth	Paramount	1952	34.00	467.05
57	Bambi	Disney / RKO	1942	102.80	462.95
60	The Jungle Book	Disney	1967	135.48	459.80
66	Peter Pan	Disney / RKO	1953	87.40	445.48

Figure 1: Screenshot of Bags executing a selection query. The data set is based on the top grossing films from IMDb. The query includes a selection predicate returning films with a release date before 1970.

almanac data as part of a research project, while at the same time gaining exposure to relational operations and database query design.

Our initial prototype is based on Snap,[2] a recent web-based environment developed by Jens Mönig at MioSoft Corporation and Brian Harvey at the University of California at Berkeley [4]. Snap is the highly interactive software featured in Berkeley's "The Beauty and Joy of Computing" course, one of the original pilot courses for AP CS Principles [7]. Harvey, Garcia, and others involved with that effort have made extensive resources available, including the Snap source code itself [3]. Although we have extended this code base to create a fundamentally different teaching tool, many of the design decisions, pedagogical features, and educational benefits share a common ancestry with Snap through its underlying JavaScript libraries.

As with Snap and the most recent version of Scratch, Bags is completely web-based and requires no additional software to be installed. The latest version and a number of educational materials are available on our website.[3] This setup not only makes it easier for us to distribute updates and enhancements to the software, but it also makes it more practical to deploy in K-12 schools. If a connection to the Internet is not available, Bags may also be downloaded in advance for offline use.

2. A QUICK TOUR OF BAGS

At a high level, Bags has a very similar layout to Snap and Scratch. Figure 1 shows the software in action. On the left we have several *palettes*, each of which consists of a set of related *blocks*. Each block represents either a data set or a relational, arithmetic, or logical operator. The user drags and drops these blocks onto the *query editor* in the center, and the corresponding *query result* appears on the right.

Suppose a student using Bags would like to look at the "Top Grossing Movies" data set and find out which of those movies were released before 1970. In relational algebra, this query would be represented as follows:

$$\sigma_{\text{Release Year}<1970}(\text{Grossing})$$

In Bags, this query is easy to formulate by using drag-and-drop functionality to create a simple selection query. The procedure of creating this query (as illustrated in Figure 1) includes the following steps:

1. Open the `Movies` data set palette, and drag and drop the `Grossing` data set block onto the query editor.
2. Open the `Relational` palette and drag and drop the `Select` operation block under the `Grossing` data set block, so that it snaps below it.
3. Open the `Operations` palette and drag and drop a "<" predicate block into the open gray circular slot in the `Select` operation block.
4. Drag and drop the *attribute block* (a drop down list that contains the current schema) into the left open slot of the "<" predicate block. Type in 1970 into the right slot of that block.
5. Click the downward arrow on the attribute block, and choose "Release Year" as the column to be evaluated.
6. Finally, click the `Select` operation block itself to execute the query. The resulting data set will appear on the right in an HTML table.

This process helps students break down a query into its component parts. Because the query design is visual, the student can see how everything fits together. It is difficult to make a syntax error because the blocks (and their slots) only snap in appropriate ways. In addition, the user may click on any block in the current query to show the result of the query up to that point. In other words, he or she can visualize the data set both before and after the selection block to understand the effect of that operation.

[2]For simplicity, we will refer to "Snap!" as "Snap" without the exclamation mark throughout the paper.
[3]`http://bags.cs.jmu.edu/`

This example is relatively trivial, but it gives students insight about how relational databases evaluate queries during their first experience with Bags. There are many other blocks available that allow students to create more sophisticated queries. The following list gives a brief summary of each palette.

- Data sets: We have compiled a diverse collection of sample data from IMDb, Wikipedia, UCI ML repository, and other websites. Users may query one of these built-in data sets or create their own from a CSV file.
- Relational: The three basic relational algebra blocks are the σ selection block, the π projection block, and the ρ rename block. There are three join blocks in Bags as well. The first is the natural join, $R \bowtie S$. The second is the theta join, $R \bowtie_\theta S$. The final is the full outer join, $R \overset{\circ}{\bowtie} S$.
- Operations: Bags also supports four set operations including union $R \cup S$, intersection $R \cap S$, difference $R - S$, and finally Cartesian product $R \times S$. The comparison operators $x > y$, $x < y$, and $x = y$ are also included, as well as the logical operations of $\wedge$ and, $\vee$ or, and $\neg$ not. Finally, there are `true` and `false` blocks that can be used in the logical operations.
- Extended: We also have blocks for γ group by, τ order by, δ duplicate elimination, and aggregate functions like `min`, `max`, `count` (and `count*`), `avg`, and `sum`.
- Attribute block: This drop down block does not relate to any one relational algebra or mathematical function. Its purpose is for the user to specify a column from the current schema. When placed near the top of a query, it corresponds to a column of the original data set. But it may also be used later in the query to refer to attributes resulting from relational operations.

Bags is free and open source software, released under the GNU Affero General Public License (AGPL 3.0). At the time of writing, we are evaluating the effectiveness of Bags as an educational tool in introductory courses as well as our upper division database course. We encourage readers to experiment with the environment, take a look at the source code, and provide feedback.

3. EXAMPLE LAB ACTIVITIES

In a classroom setting, instructors can use Bags for a variety of purposes ranging from interactive demonstrations to homework assignments. At a high level, Bags makes it easy for students to learn about relational operations in a hands-on manner. Students can also use Bags to visualize and analyze interesting data sets. This aspect of the environment is particularly applicable at the K-12 level, where the objective would not be to "learn relational algebra" but rather to develop computational thinking skills.

As a natural starting point, we recommend that students explore the built-in data sets. Figure 2 illustrates two simple queries about movies from IMDb. Notice the different elements of the each query. We start with a data set (blue) and apply a relational operation (orange) that includes a greater than operation (purple) which in turn refers back to the schema (blue). The triangle buttons on the right side

Figure 2: Introducing selection and projection using the Top 100 movies data set.

Figure 3: The relationship of Cartesian product and theta join, using the letters A–Z and numbers 1–10.

of the projection block allow the user to specify how many attributes to project from the relation.

The types of queries that students should be able to answer on day one include: "What are the top five Disney movies, in terms of adjusted gross income?" "What are the title, author, and year of books from Russia in the top 100 list?" "What are the names and symbols of the periodic elements with an atomic mass of more than 250?" In Bags, the block shapes and colors help students learn what elements are necessary for each type of query. An important concept to emphasize is that each block (or group of blocks snapped together) defines a new relation. Because of this design, students may work on multiple queries at the same time and compare their results interactively by clicking on each one.

It is often possible to design the same query in many different ways. Relational algebra provides the tools for manipulating expressions without changing their mathematical meaning. Query optimizers use *algebraic equivalences* to generate multiple query plans and select one with the minimal cost. Bags allows students to explore these types of equivalences in a visual manner by composing multiple queries and/or dragging blocks within the same query.

For example, students can visualize what it means to join two relations during a lesson with Bags. A join operation is closely related to the Cartesian product. As illustrated in Figure 3, a Cartesian product followed by a selection is essentially a theta join. By building these queries one block at a time, students can think about each step of the query and visualize what the data looks like. Using simple data sets like letters and numbers makes it easier for students to see patterns in the query results.

Consider another example query that joins "US Presidents" with "Top 100 Books" (abbreviated below as P and B). To determine which president was in office when each of the books was published, we use the join condition "P.Start $\leq$ B.Year $\leq$ P.End" (denoted by θ below). Finally, we only wish to display books published after the year 1900:

$$\sigma_{B.Year>1900}(P \bowtie_\theta B)$$

One way to rewrite this query is to "push" the selection operator inside the join as follows:

$$P \bowtie_\theta \sigma_{B.Year>1900}(B)$$

For this exercise, a student may simply rearrange the block order to confirm it produces the same result. The latter query is more efficient in practice, because it's better to filter data before sending it through an expensive join operation.

This section briefly presented some of the tasks that students and instructors might carry out using Bags. Please refer to our website for additional lab activities, homework assignments, and other resources.

4. BAGS ARCHITECTURE

The second contribution of this paper—in addition to introducing the educational tool itself—is a detailed discussion of how we designed and implemented Bags. As discussed previously, Bags makes extensive use of the JavaScript libraries underlying Snap. It is neither a fork nor a plugin of Snap, and demonstrates how Snap can be extended to create new educational environments. By documenting what we have learned in this process, we hope that others will be able to innovate educational tools in additional areas of computer science such as architecture, networking, and graphics.

4.1 Understanding Snap

Snap runs inside a virtual machine named Morphic, which is similar in architecture to how MIT's Scratch was built on top of Squeak.[4] Morphic is "a lively Web-GUI inspired by Squeak" implemented entirely in JavaScript. In order to understand how Snap works, it is necessary to become familiar with each of its main files.

- `Morphic.js` is the foundation of Snap as it creates the virtual machine and the basic objects and functions which are extended through other classes. It is based on HTML5 and supports the creation of "lively" GUI elements inside of a single Canvas object.
- `GUI.js` initializes all of the visual objects in the canvas including the toolbar, palette, scripting area, and the stage; it does not create the blocks in the palette, `Objects.js` will do this.
- `Objects.js` defines the blocks that will reside in the palette and pushes them into the GUI when a different palette is chosen and loaded.
- `Blocks.js` holds the functionality for all of the blocks that the user will create their programs with.
- `BYOB.js` provides Snap's "Build Your Own Block" functionality that allows the user to create their own custom blocks. A block editor is used that can condense a longer string of blocks into a single block.
- `Threads.js` contains the interpreter for Snap. A thread manager is used to schedule execution tasks, and each process is a *morph* that is executed an element on the Canvas object.

Morphs are the basic objects in Snap, and they can be extended into any number of submorphs. A few building block morphs from `Morphic.js` include `Color`, `Point`, `Node`, and `Morph` itself. Figure 4 illustrates the most common morphs visible to the user.

[4]See `http://www.squeak.org/`

Figure 4: Three of Snap's basic blocks.

- `BlockMorph` is the basic form of all morphs in Snap and provides a number of essential functions for all submorphs.
- `CommandBlockMorph` is the rectangle block that snaps together with other CommandBlockMorphs in order to simulate the flow of a program (by executing them in order from top to bottom).
- `HatBlockMorph` is the rectangle block with a curved top that must be at the top of a stack of blocks, usually this provides conditions for the program to execute.
- `ReporterBlockMorph` is the rounded rectangle block that is inserted in open slots in order to provide input, Booleans, mathematical functions, and many other potential uses.
- `InputSlotMorph` is a text input slot that is usually included in ReporterBlockMorphs and some CommandBlockMorphs.

As with any complex software there are many connections and interactions between each of the different files. Figure 5 summarizes the initialization process of both Snap and Bags.

Snap relies on multiple JavaScript files and at least one html file. The html file is the origin for the entire application. It creates a new WorldMorph upon loading the page. The WorldMorph is the backdrop for all of Snap. To create the WorldMorph, the html code utilizes morphic.js. After the WorldMorph is created, the IDE_Morph is created. The IDE_Morph is defined by the `gui.js` file. The gui.js file defines how the program looks and feels. This includes the palette, which holds the blocks; the scripting area, where blocks are combined to form processes; the stage; the sprite utilities (sprites are all of the images or objects in the stage); and the top of the screen including the logo, buttons, etc.

Once the IDE_Morph is created, the next file to be utilized is `objects.js`. Here, the contents of the palette are defined. There are two parts to the palette: (1) the categories (motion, control, looks, sound, etc.) and (2) the contents of each category (which has its own unique set of blocks). To create the blocks, objects.js relies on the block morphs defined in `blocks.js` (see Figure 4). Using these morphs, one can create all of the blocks in Snap. Blocks.js also defines the functions associated with each block.

The remaining files are not to be forgotten or dismissed. They all serve a function, but their function is unrelated to how we extended Snap. The design in Snap is laid out rather well, so in Bags we used a similar approach. The main differences are (1) we created our own html file, `bags.html`, and (2) we designed a change set—a group of files that contain the changes and additions to Snap. The progression of files and creation of objects works essentially the same. We did not fundamentally change how Snap works internally. Figure 6 compares how blocks work in Snap versus Bags.

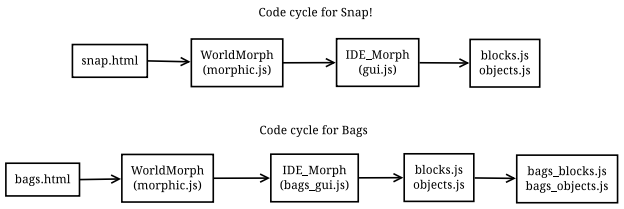

Figure 5: When Snap or Bags is instantiated.

4.2 Extending Snap

In order to extend Snap, the use of "change sets" is recommended instead of modifying the source code directly. Change sets allow one to do minimum code alternation, as well as inheriting new functionality and bug fixes as Snap is updated. To install Bags and apply our change sets, we download the latest version of Snap and extract it to its own folder. Then we add our own code in a nested folder. Our bags.html file loads Snap's JavaScript files first, and then Bags's files second in order to override some of the objects' definitions. As a result, the change sets are automatically applied without the need of patching or maintaining 3rd party code.

The overriding files for Bags include: `bags_blocks.js`, `bags_gui.js`, and `bags_objects.js`. While we did not alter every file in Snap using change sets, it should be possible to modify all of them without causing issues with updated versions of Snap.

- `bags.html` is the application entry point. After including each of Snap's JavaScript files, we include our own (e.g., first `../blocks.js` then `bags_blocks.js`).
- In `bags_blocks.js`, we edited and added new types of slots in SyntaxElementMorph's `labelPart` function. %schema adds a slot for a schema drop-down block to be snapped into. CommandBlockMorph was modified with new variables and functions to accommodate relational algebra computations.
- `bags_datasets.js` initializes example data sets (such as Top Grossing Movies). We adapted ReporterBlockMorph to provide support for attribute blocks. When it is snapped into a block connected to a data set block, it will pull in the columns in the underlying data. This file also includes all of the implementation for each of the relational algebra functions.
- The `bags_gui.js` file contains all of the changes for the GUI itself. Snap utilizes a single canvas which allows it to resize to fit nearly any window. In Bags, we currently use an html to display the data alongside the canvas. There are a number of features in Snap (e.g., cloud storage) that we have not yet implemented, so some visual features had to be removed.
- Finally, `bags_objects.js` changes the categories for the palettes and the their blocks.

As noted in the previous section, the foundation of Snap is the morph objects. It is important to utilize the basic functionality already present in Snap. Any program extending Snap will most likely be able to use the basic functionality because it is particularly robust. Snap carefully defines

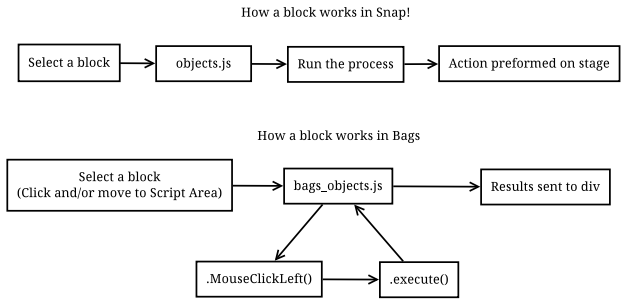

Figure 6: How a block works in Snap and in Bags.

all the interactions that can happen, i.e., how blocks snap together, but separates the behavior from the actual function of each block. This separation allows developers create the blocks they need without needing to rewrite how the program works.

Once again, we did not use custom block morphs is Bags. Instead, we found ways to take the existing morphs and alter them to suit our needs. We used functions and variables to create the desired alterations. For example, to create a data set block we re-purposed the CommandBlockMorph. We had to alter the .snap() function for this morph so that it would not allow two data set blocks to snap together—this would result in nothing meaningful in terms of relational algebra. We use a simple flag to determine whether a block is a data block. After solving the snapping issue, the next step is to assign a data set to a data block. This was accomplished by adding variables to the CommandBlockMorph. The new variables allowed us to assign a data set to a data block and pass the data set to the next block in the process.

Consider for example a user that (1) places a data block onto the scripting area, (2) connects a project block to the bottom of the data block, and finally (3) adds the drop down block to the project block. When the project block is snapped to the data block, the data set in the data block is copied to the project block. This is easily done because both the data block and the project block use the CommandBlockMorph and have the same available variables. In the case of projection, it is necessary to determine the schema that is going to be modified. Figure 7 provides a diagram of how data flows through Bags. For simplicity, we store the data set as two matrices: one is the schema and the other is the data. For each CommandBlockMorph, we added four variables for the data: schema in/out and data in/out.

We also had to introduce other flags in CommandBlockMorph to support joins and set operations. Creating these blocks required the most retooling of Snap. Unlike the other relational operations that act as filters, joins require two inputs. In other words, we need to execute two separate relational algebra functions before we could join them together. We were able to introduce a modified version of the if-else Snap block for this purpose. The top half of the block holds the left input, and the bottom half holds the right. Although the behavior of this block varies drastically from its original purpose, the design of Snap is such that we could reuse most of the code for natural join, theta join, and outer join, as well as the set operations union, difference, intersection, and Cartesian product.

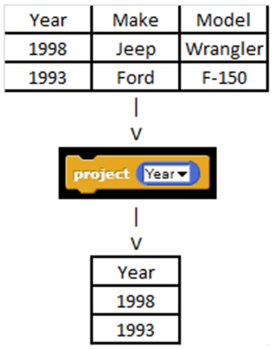

Figure 7: Illustration of a projection. Each block in Bags stores a copy of the relation (and its schema) coming in and going out of the block.

5. DISCUSSION

As previously stated in the above section, we used new functions and variables to accomplish what we needed. The reason we went this route was based on the trouble and problems we ran into when trying to create custom block morphs. The initial development of Bags was started with trying to create at least three custom block morphs. There was a DataMorph for data blocks, a JoinMorph for join blocks, and a DropDownMorph for the dropdown. The original join block was going to be a block that had two snap points on the top of the block. This was chosen because we wanted to create a top-to-bottom visualization of relational algebra (rather than left-to-right as on paper).

Implementing custom blocks in Snap is fairly easy. In an early prototype, we created custom blocks as new morphs. But as we began adding more code, we found that it caused many visual and functional challenges to spring up. Some of these included not being able to move blocks from the palettes to the scripting area, and unintentionally creating multiple instances of the block as the user attempted to release it onto the scripting area. These issues convinced us that using change sets to alter the morphs themselves was necessary to implement our desired functionality.

In addition, the custom block design had two major issues. The first was a technical issue that we were unable to have two blocks simultaneously snapped on top of a join block. Second, if someone attempted to create a long chain of joins, the process would take up a large amount of horizontal space. This design issue was deemed counterproductive, so we eventually found a new way to create the join block using the CommandBlockMorph and the if-else style block. The other morphs had similar technical and design problems that led to similar changes.

We are preparing to conduct educational research in the next stages of the project's development. One main question is whether using an interactive drag-and-drop environment changes students' perception of relational algebra. To evaluate the effectiveness of our approach, we plan to use a pre-post assessment. The pre-survey will determine prior knowledge and attitudes of the students. A control group of students will learn about relational algebra using an online tutorial by Dietrich and Goelman [2]. Another group of students will study relational algebra using the proposed environment. All students will then complete a homework assignment, followed by a post-survey. We will use the assignment to assess their understanding and the post-survey to measure changes in perception.

Bags is a robust and versatile educational tool. Just as the design of Snap allows almost anyone to learn basic computer programming concepts, Bags will open new doors to teaching students how to analyze relational data sets. We are developing a number of activities to accompany Bags and reinforce computational thinking skills at the middle school, high school, and university levels. These types of learning environments are essential for bringing new and diverse students into the computing field.

6. ACKNOWLEDGMENTS

We express gratitude to Jens Mönig and Brian Harvey for answering our many questions about the Snap code base, and for developing such a tremendous resource in the open. We would also like to thank Long Nguyen for his contributions to the initial prototype of Bags. And finally, we appreciate the thoughtful suggestions of the anonymous reviewers of this paper.

7. REFERENCES

[1] V. Barr and C. Stephenson. Bringing Computational Thinking to K-12: What is Involved and What is the Role of the Computer Science Education Community? *ACM Inroads*, 2(1):48–54, 2011.

[2] S. W. Dietrich and D. Goelman. Database Animations for Many Majors: Conference Tutorial. *J. Comput. Sci. Coll.*, 27(4):174–174, Apr. 2012.

[3] B. Harvey, D. D. Garcia, T. Barnes, N. Titterton, D. Armendariz, L. Segars, E. Lemon, S. Morris, and J. Paley. SNAP! (Build Your Own Blocks). In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 759–759, 2013.

[4] B. Harvey and J. Mönig. Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists? In *Constructionism*, Paris, France, 2010.

[5] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch Programming Language and Environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, Nov. 2010.

[6] A. Repenning, D. Webb, and A. Ioannidou. Scalable Game Design and the Development of a Checklist for Getting Computational Thinking Into Public Schools. In *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, pages 265–269, New York, NY, USA, 2010. ACM.

[7] L. Snyder, T. Barnes, D. Garcia, J. Paul, and B. Simon. The First Five Computer Science Principles Pilots: Summary and Comparisons. *ACM Inroads*, 3(2):58–60, 2012.

[8] I. Utting, S. Cooper, M. Kölling, J. Maloney, and M. Resnick. Alice, Greenfoot, and Scratch – A Discussion. *Trans. Comput. Educ.*, 10(4):17:1–17:11, Nov. 2010.

[9] U. Wolz, M. Stone, K. Pearson, S. M. Pulimood, and M. Switzer. Computational Thinking and Expository Writing in the Middle School. *Trans. Comput. Educ.*, 11(2):9:1–9:22, July 2011.

[10] F.-J. Yang. A Virtual Tutor for Relational Schema Normalization. *ACM Inroads*, 2(3):38–42, Aug. 2011.