Learning with Data
A toolkit to democratize the computational exploration of data

by

Sayamindu Dasgupta

B.Tech. Computer Science & Engineering
West Bengal University of Technology (2008)

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of

Master of Science in Media Arts and Sciences

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2012

Author [illegible]
Program in Media Arts and Sciences
August 10, 2012

Certified by
Mitchel Resnick
LEGO Papert Professor of Learning Research
Program in Media Arts and Sciences
Thesis Supervisor

Accepted by [illegible]
Patricia Maes
Alex W. Dreyfoos Professor of Media Technology
Associate Academic Head, Program in Media Arts and Sciences

Learning with Data
A toolkit to democratize the computational exploration of data

by

Sayamindu Dasgupta

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on August 10, 2012, in partial fulfillment of the
requirements for the degree of
Master of Science in Media Arts and Sciences

Abstract

This thesis explores the space of programming with data, focusing on the data-ecosystem opened up by the Internet and Cloud technologies. The central argument of this thesis is that the act of democratizing programmatic access to online data can further unleash the generative powers of this emerging ecosystem, and enable explorations of a new set of concepts and powerful ideas. To establish the validity of this argument, this thesis introduces a learning *framework* for the computational exploration of online data, a *system* that enables children to program with online data, and then finally describes a *study* of children using the system to explore wide variety of creative possibilities, as well as important computational concepts and powerful ideas around data.

Thesis Supervisor: Mitchel Resnick
Title: LEGO Papert Professor of Learning Research, Program in Media Arts and Sciences

This thesis is based on research supported by the National Science Foundation under grant numbers 1002713 and 1027848. Any opinions, findings, and conclusions, or recommendations expressed in this document are those of the author and do not necessarily reflect the views of the National Science Foundation.

Learning with Data

A toolkit to democratize the computational exploration of data

by

Sayamindu Dasgupta

The following people served as readers for this thesis:

Thesis Reader [illegible]

Harold Abelson
Class of 1922 Professor of Computer Science and Engineering
Department of Electrical Engineering and Computer Science, MIT

Thesis Reader [illegible]

Brian Silverman
President
Playful Invention Company

Acknowledgements

I would like to thank my advisor, Mitchel Resnick, for giving me the opportunity to experience this amazing adventure, and guiding me, in the most wonderful way possible, over the last couple of years. Thank you Mitch!

I'm grateful to my thesis readers, Hal Abelson and Brian Silverman, for being sources of inspiration, ideas, and for their thoughtful comments about the work I have done. A number of sections of this thesis are essentially transcripts of conversations that I have had with Hal and Brian.

The members of the Lifelong Kindergarten group have been amazing—a special note of thanks goes out to Chris Garrity for arranging and helping me with the workshops in the after-school Scratch club, and to John Maloney for being incredibly patient and helpful with all my last minute code changes and requests. Also, each and every member of LLK + extended family has contributed to this thesis in one way or the other, so a huge thank you to Abdulrahman Y. idlbi, Amos Blanton, Amon Millner, Andrés Monroy-Hernández, Champika Fernando, Claudia Urrea, Eric Rosenbaum, Evelyn Eastmond, Jay Silver, Karen Brennan, Michelle Chung, Natalie Rusk, Paula Bonta, Ricarose Roque, Stephanie Gayle and Tiffany Tseng. As we say in LLK, "very good, very good, yay!".

The Media Lab and MIT has been an amazing place to be in, with a wonderful community, and I thank all my friends at the Lab and the Institute who have been sources of joy and encouragement.

Thanks to friends outside of Cambridge as well, especially to Dipankar Das, Sankarshan Mukhopadhyay and Susmit Shannigrahi for reading drafts of the thesis and offering valuable comments.

A huge thanks to Sutanuka Roy, for all the fun conversation and chats, and for occasionally helping me maintain my sanity.

And last, but not least, thanks and love to my family, especially to my mother, for being my inspiration, and always encouraging me to do the things that I love to do and care about.

Contents

List of Figures

"Sixty years ago, digital computers made information readable. Twenty years ago, the Internet made it reachable."

Chris Anderson [1]

1 Introduction

The world is going through a data-explosion. Today, in a time-span measured in days, we generate an amount of data that surpasses the amount of data previously created over the course of entire human history [2]. This explosion is largely made possible by lowering cost of data storage, but what ultimately makes all this data useful is easy access. This access is made possible by the Internet, and more so, by the recent emergence of connected, on-demand computing resources, something that is often referred to as the "Cloud" [3]. Anyone with a connection to the Internet today has access to an incredible amount of storage space and computing power, thanks to services like Amazon Web Services, Google App Engine, Microsoft Windows Azure, etc.

In this scenario, the model of computing is fast changing to a more distributed model—as described in a recent *The Observer* article titled "A manifesto for teaching computer science in the 21st century", there is a shift from the world wide web to the "world wide computer" [4].

> "Understanding how an individual machine works is only part of the story. We are rapidly moving from a world where the PC was the computer to one where "the network is the computer". The evolution of "cloud computing" means that the world wide web is morphing into the "world wide computer" and the teaching of computer science needs to take that on board."

This changing model, along with the lowered barrier to data access and exploration raises interesting new possibilities for computer science education and beyond. We now have the possibility of systems that are always connected to repositories of data, and as a result of this, introductory lessons in the computational exploration of data need not be restricted to spreadsheets or "toy datasets" or pen and paper surveys anymore. Programs designed and developed by novice programmers can interact with globally connected, centralized data repositories of various sizes, enabling exploration of a wide range of concepts. Some of these concepts include novel methods of collecting data and building datasets (e.g. through crowdsourcing), representing datasets (eg: with interactive visualizations), analyzing datasets (eg: through map-reduce), etc. Moreover, with the global reach of the Internet, even comparatively simple data-based tools such as surveys become much more compelling and authentic, as one can reach out to a much larger, geographically diverse community by hosting the survey front-end and the underlying datastore online.

In this thesis, I set out to explore the space of programming with data, focusing specifically on the generative area that is made possible by the data-ecosystem opened up by the Internet and Cloud technologies. I argue that by bringing programmatic access to online data to a broader audience, we can further unleash the generative powers of this emerging area, and enable the explorations of a new set of concepts and ideas.

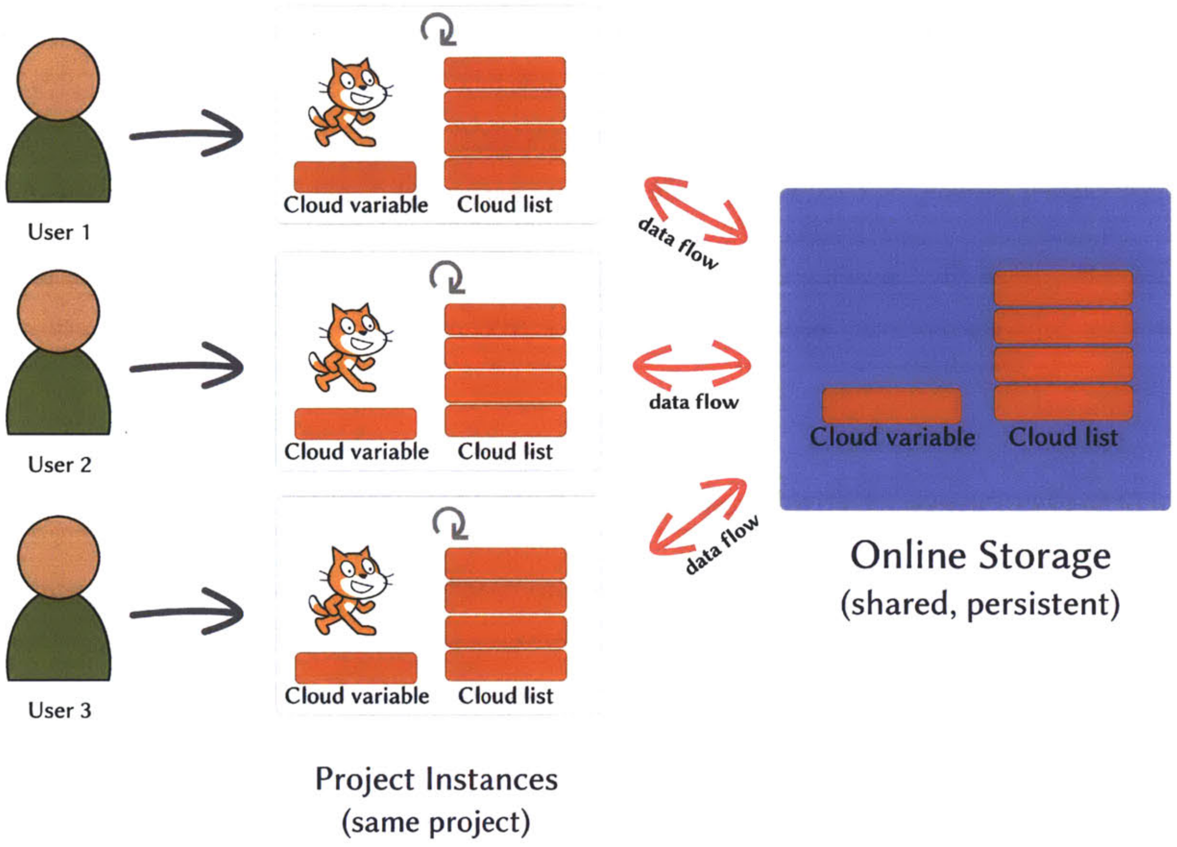


Figure 1-1: *Overview of Cloud data in Scratch 2.0. The value of Cloud data-structures are stored online in a server persistently, and parallel instances of a Scratch 2.0 project share the values*

1.1 A Toolkit to Program with Online Data

To achieve this goal of democratizing programming with online data, I build a set of online-data extensions for Scratch 2.0, an existing programming toolkit for novice programmers. The extensions add an orthogonal property to Scratch 2.0 data-structure instances (variables and lists) which is used to determine the storage mechanism for the data-structure instance—if the property is set, apart from being stored in memory during runtime, the value of the instance is stored online persistently[1]. Having an online storage mechanism also leads to the data-structure value being shared across parallel instances of the project (figure 1-1). Building on the distinction drawn up

[1]Scratch 2.0 itself is hosted online, with instances of Scratch 2.0 projects running in the web-browser. This makes it easier to have an online storage backend for Scratch 2.0 data-structures.

by diSessa [5], the extensions are *structurally simple*, with a small incremental change in the overall characteristics of the data-structure, but at the same time, open up a large number of *functional possibilities* and use-cases. To the Scratch 2.0 programmer, these data-structures are presented as a new category of variables and lists, and are called *"Cloud variables"* and *"Cloud lists"* respectively.

Following up on the creation of the extensions, I show, as a part of the evaluation of my work, how these comparatively simple additions lead to a very generative space for a diverse set of creative applications.

1.2 Overview of the Thesis

In Chapter 2, I describe the idea of a generative platform and give an outline of how programming with online data, and the act of utilizing the affordances of data-ecosystem in the Internet can be a generative space, leading to numerous possibilities for creative activities and explorations. I present a list of possible creative uses of programming with online data, and conclude by defining my research focus, along with a listing of the contributions of this thesis.

In Chapter 3, I layout the pedagogical framework for my work, connecting to Seymour Papert's idea of constructionism, as well as to the concept of powerful ideas. In this chapter, I also present four core computational topics around data, along with the powerful ideas inherent in these topics, as a framework that can be used to explore programming with data. Each idea is a combination of computational concepts, as well as practices, and I unpack each idea into its constituents in this chapter.

In Chapter 4, I enumerate the design inspirations behind my work. I describe some of the current practices in the Scratch community that helped inspire and guide my design, and also conduct a survey of prior systems which have similar goals and/or functionality. For this survey, I cover traditional programming language research, as well as visual programming language systems designed for novice programmers.

In Chapter 5, I introduce my final design, beginning with my design goals and guidelines, and culminating in the actual description of the design. I also include a list of the implications of my design in the context of the larger Scratch community, along with descriptions of my responses to these implications.

In Chapter 6, I describe six Scratch 2.0 projects that illustrate the diversity of creative possibilities enabled by Cloud data-structures. These projects are outcomes of user-workshops, as well as my own creations, intended to be samples.

In Chapter 7, I describe a number of user-studies of the system, in various settings (after-school club, online with beta-testers, workshop, etc.). I focus on the learning outcomes, confusions and misconceptions and resultant computational perspectives. I conclude with a short exploration of the mental models that the users form for the entire system.

In Chapter 8, I conclude by exploring possible future directions in the space of programming with data, with a focus on topics such as programming for mobile devices, accessing Open Datasets, the representation of data, etc.

Additionally, a technical description of the implementation details of the Cloud data-structure system, along with a listing of the technical challenges for such a system is included in this thesis as Appendix A.

"If the personally owned book was one of the main shapers of the Renaissance notion of the individual, then the pervasively networked computer of the future should shape humans who are healthy skeptics from an early age."

Alan Kay [6]

2 Computational Inspirations

In this chapter I describe the concept of a generative platform, and give an overview of how programmatic exploration of online data can become a generative platform for a variety of possibilities. I also qualify the focus of my thesis further, and try to pinpoint the boundaries of the space I'm trying to explore.

2.1 The Generative Grid and a Generative Platform

Our digital lives are increasingly being mediated by the Internet and Cloud technologies. Fifteen years ago, the Internet served primarily as a communication channel. Today, apart from being the conduit for our digital communications and social interactions, the Internet is fast becoming *the* storage repository for data created by us on a day-to-day basis. This includes both the data we explicitly create, as well as data we generate through our browsing patterns, social interactions, etc. While at present this is more true for mobile devices, the trend is quickly becoming universal for all kinds of computing devices—the increasing popularity of cloud-based services and applications such as Google Docs being an illustrative example. Of course, this model is nothing new, and a strikingly similar utility based ecosystem was predicted in 1961 by John McCarthy in a talk entitled "Time-Sharing Computing Systems" [7].

> "We can envisage computing service companies whose subscribers are connected to them by telephone lines. Each subscriber needs to pay only for the capacity he actually uses, but he has access to all programming languages characteristic of a very large system."

However, it is important to recognize here that the power of the Internet and Cloud technologies goes much beyond being a storage space and/or a transfer conduit for data and meta-data. The emergence of the Internet and the Cloud has led to not only a shift in *technological infrastructure*, but also a change in the *technological mindset*, as described in *The Observer* article referred to in chapter 1. The affordances of the Internet and its ubiquity has been harnessed by millions of people all over the world for various novel and creative applications, from giants like Facebook to small hobby projects like tools that let one control a coffee-maker from across the continent. In 2006, Jonathan Zittrain used the term "generative" to describe the Personal Computer (PC) and the Internet [8]. He described generativity of a technology platform as its

> "[...] overall capacity to produce unprompted change driven by large, varied, and uncoordinated audiences."

Zittrain then went on to point out the thinning boundary between the PC and the Internet, and their fusion into a "generative grid":

> "Significantly, the last several years have witnessed a proliferation of PCs hosting

> broadband Internet connections. The generative PC has become intertwined with the generative Internet, and the whole is now greater than the sum of its parts. A critical mass of always-on computers means that processing power for many tasks, ranging from difficult mathematical computations to rapid transmission of otherwise prohibitively large files, can be distributed among hundreds, thousands, or millions of PCs. Similarly, it means that much of the information that once needed to reside on a user's PC to remain conveniently accessible — documents, e-mail, photos, and the like — can instead be stored somewhere on the Internet. So, too, can the programs that a user might care to run."

This generative grid (albeit with a less peer-to-peer structure than the above description[1]), combined with online data gives rise to a generative platform, enabling opportunities for diverse explorations, through programming, of various areas such as social systems, personal data analytics applications, civic tools, etc (examples of such systems are enumerated later in this chapter).

2.2 Bringing Ease of Mastery to the Generative Platform

According to Zittrain, "ease of mastery" is one of the defining characteristics of a generative system [10]. He qualifies the phrase further by defining the word "ease" as something with which people can get started with a given technology or tool, using the example of a pencil [8]:

> "Handling a pencil takes a mere moment to understand and put to many uses even though it might require innate artistic talent and a lifetime of practice to achieve da Vincian levels of mastery with it. That is, much of the pencil's generativity stems from how useful it is both to the neophyte and to the master."[2]

My argument is that in-spite of great creative potential, this form of ease of mastery is a missing piece in the context of the generative platform of programming with online data. There is a high

[1] A centralized grid can be problematic for generativity, as described in by Zittrain in another, more recent article [9].

[2] This also is reminiscent of Seymour Papert's concept of "hard fun", where the "hardness" is determined by factors and needs that are relevant and are not extraneous [11].

barrier to entry for beginners who want to program with online data, and the process requires the mastery and understanding of a number of complex server side technologies, as well as client side systems. For example, to develop a simple online message-board, one needs know a server-side programming language such as PHP, be familiar with a database management system such as MySQL, and be conversant in client side technologies such as HTML and Javascript. Apart from being familiar with all these individual technologies, one also needs to have an overview of how all of the above interact with each other, and more often than not, some basic understanding of a server-side Operating System such as GNU/Linux is also required.

If this considerable barrier to entry is lowered, my hypothesis is that we can expect to see the emergence of greater generativity and creativity. This hypothesis is driven by what Mitchel Resnick and Brian Silverman describe as the "low floor" and "wide walls" design principle for making construction kit for children [12].

> "...features that are specific enough so that kids can quickly understand how to use them (low floor), but general enough so that kids can continue to find new ways to use them (wide walls)."

2.3 Creative Possibilities—Leveraging the Generative Platform

Through my thesis, I implement ways in which this "ease of mastery" can be brought within the reach of a broader population of programmers. My aim is to enable novice programmers to express themselves in a wide range of ways on by leveraging the generative platform of programming with online data. Some of the possible expressive artifacts that can be achieved through programmatic access to online data are described below[3]:

[3] A question here may be, what is the value of implementing these systems where similar tools may already exist in pre-packaged form? The answer lies in the sense of empowerment an young programmer has on being able to build and customize something he cares about at a personal level, as well as at a larger, social level. I describe this in terms of relevance and authenticity in the next chapter (section 3.1).

Open Data

A soccer fan wants to create a tool to find the most consistent performer in his favorite soccer team across different soccer world cup tournaments. He gathers data from Wikipedia, and builds a tool to analyze it.

Over the past few years, "Open Data" initiatives have published datasets about a wide range of topics—ranging from climate data to detailed statistics about famous sports teams. These datasets are hosted online, can be freely used by anyone, and in most cases, can also be remixed by anyone to create compelling "data mashups", as illustrated by the user-story above.

However, the Open Data is only a part of the story. Another affordance of the Internet and the Cloud is to act as a centralized repository of user-created data, as the use-case story below shows:

Crowdsourcing

A student in India wants to understand the patterns behind "power-cuts" (unplanned loss of electrical power—a serious problem in most developing nations). She builds an online tool for her social-network friends from all over the country to report power-cuts as they happen in their localities. With this data, she creates a heat-map visualization that shows the areas worst affected by the problem [4].

The act of crowdsourcing [13], where one can "farm out" small tasks to others over the Internet, can be made almost effortless with a toolkit that allows for easy-to-use programmatic access to the online datastores.

Personal Data Analytics

A long distance runner wants to get a deeper understanding of his running. He creates a mobile app to log accelerometer data as he runs, which he later processes to understand the

[4]See http://powercuts.in for a real world implementation of this idea.

variation in his rate of steps. He then shares it online through a custom web application for other members of his running community to see and comment on.

User-contributed data need not be always crowdsourced—with the ubiquitous presence of sensors in the modern world, especially through smartphones, it is possible to collect data from a single source about a specific parameter over a period of time, store it online for subsequent retrieval, analysis and sharing.

Social Systems

An young programmer creates a chatroom system where he and his friends can plan out their next programming project. The chatroom client uses the webcam of the user's computer to detect whether the user is active or not, and marks the user as inactive in the chat if no movement is detected within a specific amount of time.

The Internet provides a communication medium and a centralized data-repository to its users, no matter where they are situated on the globe. For an online tool, an user in India accesses the same datastore and processing system as those accessed by a user from Peru. This makes it an ideal foundation block for building computational social systems.

Civic Systems

A teenager wants to create an event advertisement system for her neighborhood. She implements an online message-board where others can post news about upcoming events, and based on feedback from friends, adds a feature where event attendees can post their reflections and thoughts after the events.

As with social systems, tools with a civic agenda can also be created with online data. These tools can range in complexity from simple surveys to sophisticated message boards where communities can come together, advertise local happenings, etc.

2.4 Research Focus—Programming with Data

For this thesis, my primary focus and goal is to explore how non-expert programmers can build systems that enable them to collect and access data. In this section, I further qualify the goals and scope of my work, drawing boundaries between what I cover and what I leave out for future explorations.

The computational exploration of data can be broken up roughly into four areas—building data(sets), processing data, representing data(sets) and organizing data [5] (this entire space is described in detail in Chapter 3, section 3.2). For this thesis, I focus only on a subset of these areas.

Apart from the four areas describe above, there is also a distinction between "designing with data" vs. "gaining insights from data". The distinction is largely dependent on how a given creative process or act is situated – if data is a part of a larger game, or a story, or if it is represented in a novel manner (e.g. pitch of an audio sound representing magnitude of data-points), the design element is brought into the forefront. The focus of this thesis is more on "learners as designers" rather than "learners as scientists", though nothing prevents the toolkit from being appropriated in ways that are closer to the data-science end of the spectrum.

With these caveats in mind, the core contributions of this thesis are:

- A learning *framework* for the computational exploration of data as mediated by the Internet, with a list of topics that cover the entire space, along with certain "powerful ideas" that emerge out of these topics.
- The design and implementation of a *system* that allows non-expert programmers to build systems that can store, access and share data online.
- A *study* of children using the system, looking at the learning outcomes, which includes understanding of topics and ideas from the framework, as well as confusions and misconceptions arising out of the system and its design.

[5] I use the term dataset to mean a collection of data, and the term database to refer to not only the data, but also the infrastructure to manage, store and retrieve the data.

"Why then should computers in schools be confined to computing the sum of the squares of the first twenty odd numbers and similar so-called 'problem-solving' uses? Why not use them to produce some action? There is no better reason than the intellectual timidity of the computers in education movement, which seems remarkably reluctant to use the computers for any purpose that fails to look very much like something that has been taught in schools for the past centuries."

Seymour Papert & Cynthia Solomon [14]

3 Pedagogical Inspirations

In this chapter, I describe the pedagogical inspirations and background of my work, and develop a framework to think about the computational exploration of data.

3.1 Construction, Relevance, and Authenticity

Many of our best learning experiences happen when when we are actively engaged in designing and creating artifacts, especially ones those are personally relevant and are situated in an authentic, real-world context [15]. This forms the cornerstone of the pedagogical approach of this thesis.

Construction of Digital Artifacts

Scratch 2.0, the programming toolkit that I build upon, follows a long tradition of digital construction kits, and the underlying principle behind these toolkits is the theory of *Constructionism* put forward by Seymour Papert. Constructionism builds upon the theory of *Constructivism*, which describes learning as "building knowledge structures". To this concept, Constructionism adds the idea that the knowledge-structure building occurs "especially felicitously in a context where the learner is consciously engaged in constructing a public entity" [16].

These acts of construction can happen in any context, and in his 1980 book *Mindstorms* [17], Papert talks about his own experience as a child where playing and building with gears led to an intuitive understanding of abstract but powerful mathematical concepts such as ratios and multi-variable equations. However, at the same time, Papert points out that computers have their own special, privileged place in Constructionism as they "provide an especially wide range of excellent contexts for constructionist learning." This becomes especially true for situations that involve programming computers, as the act of programming brings out the true universality of computers, catering to thousands of tastes, interests and passions.

With the data-extensions that I implement for this thesis, my aim is to enable the construction (through programming) of digital artifacts that involve data. Through the acts of digital construction with data, learners would be able to explore data-related concepts and ideas in a multitude of ways, thanks in part to the universality of computers, and in part to the generativity that characterizes programming with online data (as illustrated in the previous chapter).

Relevance, or, Connecting to Personal Interests

In *Mindstorms*, Papert also points out that in order to engage learners, learning activities and the acts of construction need to be personally relevant and meaningful. He highlights the importance of connecting learning contexts to the interests of the learner with an example:

"In the LOGO environment new ideas are often acquired as a means of satisfying a personal need to do something one could not do before. In a traditional school setting, the beginning student encounters the notion of variable in little problems such as:

$5 + X = 8$. What is X?

Few children see this as a personally relevant problem, and even fewer experience the method of solution as a source of power. They are right. In the context of their lives, they can't do much with it. In the LOGO encounter, the situation is very much different. Here the child has a personal need: To make a spiral. In this context the idea of a variable is a source of personal power, power to do something desired but inaccessible without this idea."

However, in spite of Papert and his colleagues pointing this out as a problem as early as in 1971 [14], a lot of the standard programming exercises for teaching children programming even today consists of tasks such as generating a list of prime numbers, or the Fibonacci series. While these exercises are valuable, and highlight important computational concepts and mathematical ideas, they appeal to only a very small percentage of young learners [18]. In the space of computational exploration of data, the situation is largely similar, with a lot of introductory activities remaining restricted to very generic "problems", and most of these tasks have little or no personal relevance or meaning for the learners.

To address this deficiency, the extensions that I implement for Scratch 2.0 place the act of exploration of data within the larger context of a Scratch project. This project may be a game, a story, a simulation, a survey, or a novel way of representing data through visualization or similar means, catering to multiple interests and tastes. Thus, following the pedagogical principle set forth by Papert, the exploration of data becomes necessary for "satisfying a personal need". As examples, understanding how to collect, organize and retrieve data becomes a part of maintaining a high-score list in a game, or keeping track of the choices made by a user in a choose-your-own-adventure story. Of course, if someone wants to explore data for its own sake, no one is preventing that from happening, and one may argue that building Scratch projects for tasks such as collecting data through simple surveys, and then visualizing them is closer to what introductory computation+data-oriented courses have been traditionally doing, but with added personal relevance and meaningfulness.

Authentic Experiences

Towards the beginning of the twentieth century, educators and thinkers like Dewey [19] and Tagore [20] started to point out the importance of connecting the real world to learning experiences of children in formal learning environments. As learners get involved in projects situated in communities and contexts they care about, they become more engaged and find the entire experience more meaningful and authentic [21; 22]. While Papert describes personal relevance (as described in the previous sub-section) through the lens of what he terms as the "power principle" (i.e. empowering the learner to perform personally meaningful projects), this idea of being situated a larger social context is described as embodying the "principle of cultural resonance" [17].

Online-data extensions bring the scope of certain types of authentic experiences into the realm of Scratch via the possibility of building social and civic systems with the data-extensions. As described with examples in the previous chapter, these systems enable the young programmer to reach out into the real world over the Internet, solicit input and participation from a real community (which can be the community which the programmer belongs to), and create a virtual space for computer-mediated interactions. In the end, all of these come together to lead to a more powerful sense of accomplishment and "impact", with significant "cultural resonance", which can only help as a motivator for an engaged learning experience.

3.2 A Framework for the Computational Exploration of Data

Moving from larger, more general pedagogical principles, in this section, I enumerate four topics that span the space of computational exploration of data. For each topic, I unpack it into a set of concepts and practices. Within each topic, I also try to tease out inherent *powerful ideas*, where each idea, in Seymour Papert's words, is "general" (i.e. applicable over a large domain), "intelligible" (i.e. easy to grasp), and "personal" (i.e. rooted in experience) [17]. Though powerful ideas can be "tools to think with over a lifetime", learning frameworks used in formal educational settings often tend to overlook them [23], and hence I make an attempt to explicitly point out the ideas which are worth exploring within a given topic. The topics that I describe are:

- Building Data
- Processing Data
- Representing Data
- Organizing Data

Computational Topics around Data

I start off with a couple of stories that involve programming and data:

Game with a high-score list: A young programmer wants to keep track of all the scores in a game that she has created with Scratch 2.0, along with the usernames of the people who attained those scores. Initially she uses a single Cloud list[1], with each item in the list being a score concatenated with the corresponding username. However, she discovers that this leads to problems later on, when she tries to sort the list and get the username with the highest score. Then she decides to use two Cloud lists, one for storing scores, and the other for storing usernames. She writes some extra code in the score-collection system to ensure that the position of the score and the scorer in each list remains in sync and sorted. This solves her high-score retrieval problem.

Weather "visualization" app: A young programmer collects rainfall predictions for various latitudes and longitudes from the national weather database. He creates an interactive map application where taking the mouse cursor over a particular location would trigger a audio sound playback—if the prediction is for good weather, the sound is of birds chirping, if there's rain in store, the sound is of raindrops. Internally, he uses a key-value data-store to store the daily prediction information.[2]

[1]List is the only compound data-structure available in Scratch

[2]This project was inspired by Music Bottles [24], developed by the Tangible Interfaces group, MIT Media Lab.

Topic: Building data

A number of things are happening in the first story above. There is a process of iterative design of the dataset "schema" and also the code for collecting the scores involves sorting as well as synchronization between two data-structures. Often, all of these practices are bundled into the concept of "data-collection" [25]. However, as the story illustrates, it is quite clear that the word "collection" does not capture all these activities–it is merely one activity within a larger set. Hence my argument is that in place of "data collection", the concept should be called "building data(sets)" instead.

At a higher level, in terms of real-world practices, especially in the context of online data, two techniques are often used for building data:

Crowdsourcing: Crowdsourcing is usually defined as the practice of solving a computational task by dividing it into small chunks, and then distributing them to a network of people ("the crowd") who solve each subtask as an individual task in a relatively short amount of time [13]. In the specific context of data, crowdsourcing can take the form of collecting information from a diverse group of people. The collected data can range from votes about a given topic to user-submitted reports on topics such as crime, power shortage (as described in Chapter 2), etc, and the context of crowdsourcing can also vary from being restricted to a particular community (e.g. all the students of MIT) to being open to the entire world.

Logging: Logging is the practice of collecting data from a given source over a period of time. An emerging scenario for logging is personal data analytics (described in Chapter 2), but it has traditionally been extensively used to monitor systems that run over extended periods of time.

The attributes of powerful ideas are present in both crowdsourcing and logging, perhaps more so in crowdsourcing than logging. However, another very powerful idea that emerges in the space of building data is that of **privacy**, something that gives one a framework to think about *what* data is being collected, and also *how* it is being used. Privacy has far-reaching relevance in the present age of collecting data about everything and anything, and in Chapter 7, I discuss how the question of collecting data with Cloud data-structures led children to think about privacy in a variety of ways.

Topic: Processing data

In our first story the programmer has to ensure that the data is sorted, and in the second story, the programmer writes code to perform search on the dataset to get the correct results. Moreover, in the first story, the program has to identify the high scores when displaying the scoreboard at the end of a game session (which also involves sorting, though in this case, it is done beforehand).

Sorting and searching are the more basic types of processing or analysis tasks that can be a done on a given dataset, and they often constitute the building blocks for more complex tasks (which in turn, are usually concerned with improving the signal-to-noise ratio of the data, and bringing out only the relevant data-points). However, if we drill down further, for most processing or analytics tasks, the low-level operations are usually a combination of

- a *map* operation where a function is applied to each element in the dataset. (e.g. squaring all the elements of a list of numbers)
- a *reduce* or a *fold* operation where a function is applied cumulatively to the elements in a dataset, to produce a single value in the end. (e.g. adding all elements of a list of numbers)
- a *filter* operation where a function is applied to each element in a dataset, and the result is a new dataset which has all the elements for which the function returns true. (e.g. checking if a given element of a list of numbers is positive, and creating a new list with only the positive numbers)

Though these tasks often form a part of data analysis problems, they can be useful elsewhere as well (e.g. while storing data, as in the first story, or while retrieving data, when filtering is a common subtask). Hence, instead of labeling this topic as "data analysis", I term it as "processing data".

A powerful idea inherent in this topic is the phenomena of how very complex operations can be broken down into a limited set of simpler tasks. For example, in the present context, a significant percentage of data-processing tasks can be decomposed into various combinations of the basic operations listed above. This was demonstrated in 1979 by Richard Waters with an analysis of a number of Fortran programs from the IBM Scientific Subroutines Package, where he found that a significant percentage of the program loops could be represented by variants of the map-reduce-filter structure [26].

Topic: Representing data

In our second story, data is presented to the end user in a novel way through the aural medium. Representing (or summarizing) data is often a key goal for a significant number of data-related projects. Most of the representation usually happens through aggregation operations on data, or through visualizations:

Aggregation: Aggregation involves extracting some information out of a dataset (usually in order to provide a summary)—traditionally, it has been implemented through methods such as deriving the mean, median, etc. Most of these practices and methods draw from existing fields such as statistics, however, in the context of computation, aggregation may also cover novel operations such as geospatial queries (e.g. finding the *n* closest objects near a given latitude-longitude pair), etc.

Visualization: Using visual images to represent data has been an established practice since the late 18th century [27]. However, computational data visualization can deal with datasets that are orders of magnitude larger than what was previously manageable through traditional analog (and largely manual) methods, and can incorporate multimedia elements, as well as interactivity, like the weather app example mentioned earlier in this chapter. Often, there is a strong element of design involved in the creation of a visualization, and this intersection of design and computational data was covered by Benjamin Fry in his PhD thesis dissertation, "Computational Information Design"[28].

Again, within this topic, both aggregation and visualization provide numerous pathways to powerful ideas—e.g. the idea of gauging the "spread" of a dataset from the standard deviation, or the idea of representing numeric quantities through spatial attributes (e.g. in a graph), etc.

However, these forms of data representation are external—in each case, it is usually what the end user of a given program gets to see. There is another aspect of data-representation, which happens internally within a given program, and is achieved by choosing a particular type of data-structure. That falls within the space of data-organization, which is the described below.

Topic: Organizing data

In both the stories above, one of the common underlying questions concerns the organization of the data. The first sub-problem in this question is *if* the data needs to be organized, as in a number of data-related computational problems, there is a trade-off between caching data and computing it on-demand. For example, in the first story, the data is pre-sorted, though there is the alternate possibility or strategy of sorting the data only when needs to be displayed.

The second (and often larger) sub-problem is *how* the data is to be organized (at a lower level than the "schema"). Different strategies of organizing data at a "under-the-hood" level, or in other words, different data-structures have their own, unique set of advantages and drawbacks, and an incorrect choice of data-structure while designing a program can lead to complications in later stages of program development.

A powerful idea inherent in organizing data is the **abstraction of data**, something that, as described in the *Structure and Interpretation of Computer Programs*, "enables us to isolate how a compound data object is used from the details of how it is constructed from more primitive data objects" [29]. Thus, with Scratch 2.0 lists, one can write a set of procedures that treat a two-element list as a pair of coordinates, or as a point on the Cartesian plane. These procedures can be used to calculate distance between two points, draw lines between two points, draw polygons between more than two points, etc. However the power of the idea inherent in this act lies in the fact that while using these procedures, the programmer does not have to think about the underlying construction of the point data-structure, the underlying details are abstracted away.

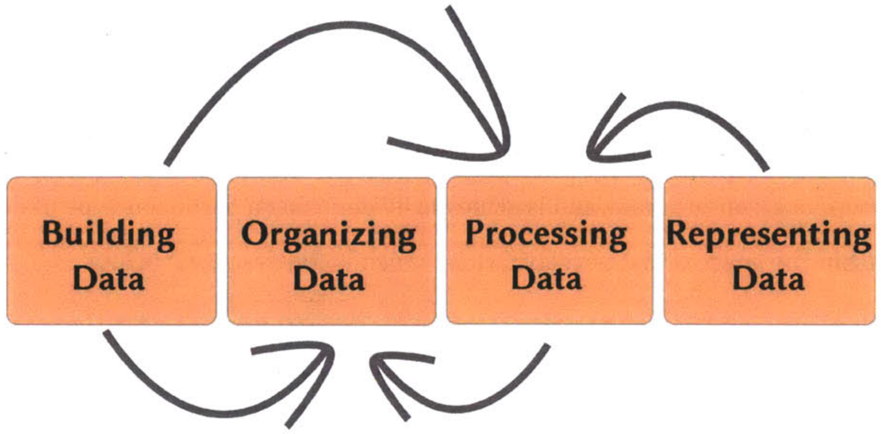


Figure 3-1: *Topics for computational explorations of data, along with their inter-dependencies (e.g. data-representation often depends on data-processing)*

Before concluding the enumeration of computational topics around data, I should point out as a caveat that the boundaries between the topics are not always very sharp—building data can draw upon concepts from organizing data, data-representation can draw upon data-processing and so on (figure 3-1). However, as far as the basic computational topics around data go, these do a good job of covering the entire space, at least at the introductory level. Hence, covering the above concepts and practices can form the core of a series of activities designed to explore computation in the context of data. The extensions that I design as a part of this thesis enable young learners to explore at least a subset of these topics, while simultaneously, connecting back to the principles of construction, personal relevance and authenticity that are enumerated in the first half of this chapter.

"[...] creation always involves building upon something else. There is no art that doesn't reuse."

Lawrence Lessig

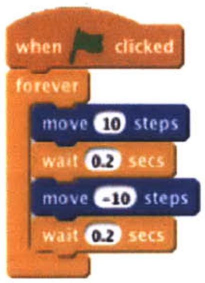
4

Design Inspirations

In this chapter, I describe the design inspirations for the extensions that I built. The inspirations arise from practices in the community,as well as from existing systems and designs that helped form my final design and implementation. I begin by describing the underlying programming toolkit that I use as the base of my work, and the community around the toolkit, in order to provide the required context and background information.

4.1 Overview of Scratch

Scratch is a visual, block-based programming language and environment for young programmers [18]. Scratch programs consist of graphical characters (called sprites) on a stage, and each sprite's behavior is determined by one or more stack of visual instruction blocks (figure 4-1).



```
when [flag] clicked
forever
    move 10 steps
    wait 0.2 secs
    move -10 steps
    wait 0.2 secs
```

Figure 4-1: *Scratch block stack to make a sprite move back and forth*

Scratch also has an associated online community website (figure 4-2) where members can upload programs, remix programs created by others and comment on and bookmark each other's programs [30]. As of June 2012, the community has more than a million registered users and more than 2.5 million uploaded programs.

Scratch | Home | Imagine, program, share

home projects galleries support forums about Language

Login or Signup for an account

Create and share your own interactive stories, games, music, and art

Check out the 2,639,523 projects from around the world!

To create your own projects:

Download Scratch

Featured Projects See more

Solving the Tow... by mathsboy314

The Hunger Game... by mcectrox

Crane race. Exc... by eventexception

How do you use tags?

Tags: games, stories, animations

We want to learn more about how Scratchers use tags. Take our brief survey!

View survey

ScratchEd

Do you help people learn Scratch? Join ScratchEd, our new online community for educators.

Projects Selected by Lunastar- Learn more

Figure 4-2: *The Scratch community website*

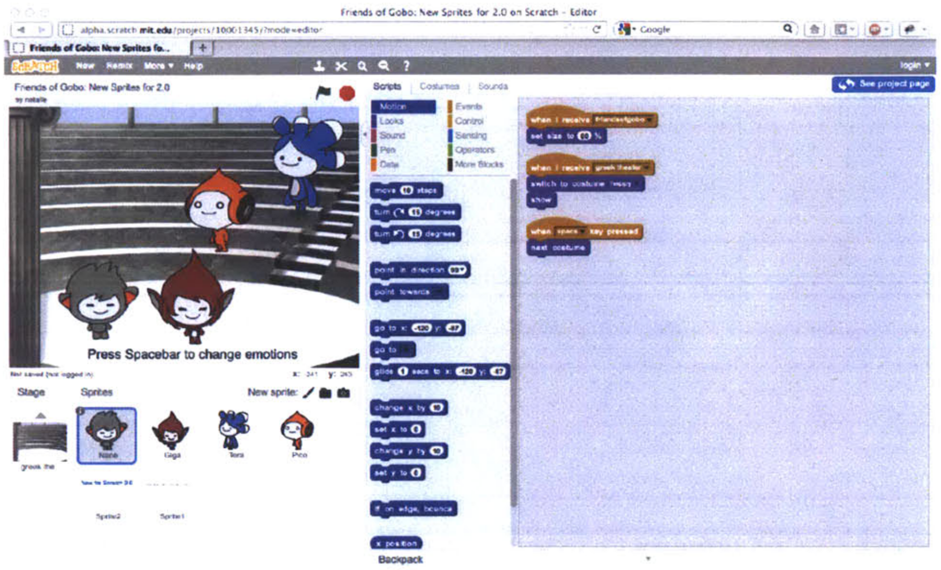
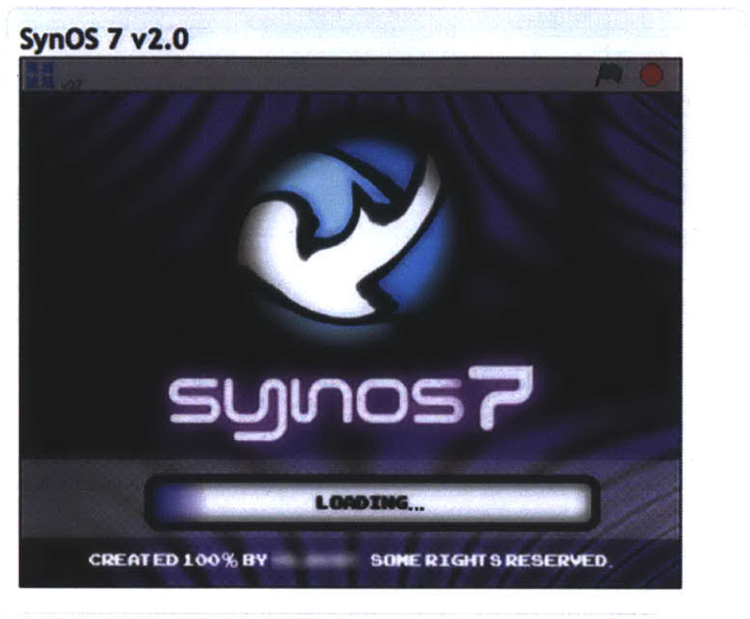


Figure 4-3: *Scratch 2.0 editor in the Firefox web browser*

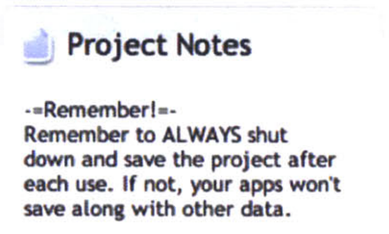
The next major release of Scratch, Scratch 2.0, is going to be hosted online, so that instead of having to download a standalone Scratch authoring tool, young programmers would be able to directly create programs in the web-browser (figure 4-3). These programs will also be hosted on the Scratch 2.0 website, and would be executed directly on the web-browser as mini web-apps.

4.2 Existing Scratch Community Practices

Scratch 1.x programs run in a sandboxed environment, with limited access to system resources like the file-system, hardware devices etc. There is no persistent storage support in the Scratch 1.x language design – data generated during the execution of the program is ephemeral. However, this has not prevented Scratch programmers from using various workarounds to store data in Scratch programs. I discuss two common categories of projects using these workarounds that motivated my work—Scratch "operating systems" and Scratch survey projects. A third community practice of "hacking" into Scratch and enabling hidden features also inspired my work, and I describe it in this section as well.



(a)


Project Notes

-=Remember!=-
Remember to ALWAYS shut down and save the project after each use. If not, your apps won't save along with other data.

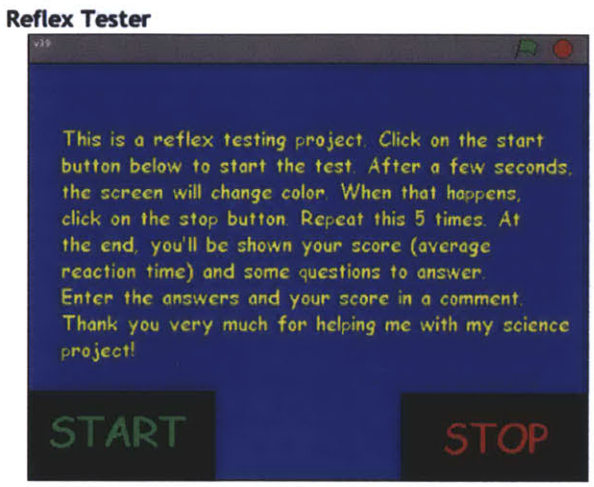
(b)

Figure 4-4: *An "operating system" project in Scratch, along with notes reminding the user to save the entire program after a session*

Scratch "operating systems"

Values in Scratch 1.x data-structures are stored along with the program code when a Scratch program is saved to disk. This was done in order to enable functionality similar to that provided by constants in traditional programming languages (e.g. storing the value of π). Programmers using Scratch have taken advantage of this feature in various ways.

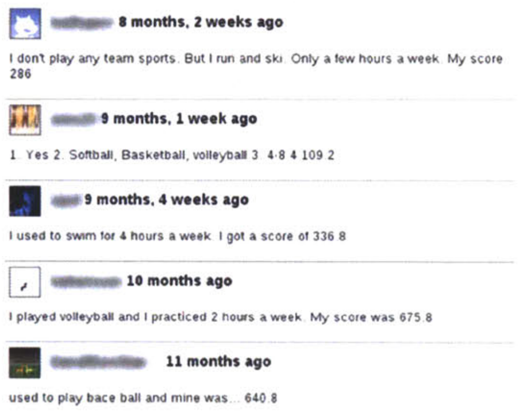
A typical example in this category are toy "operating system" programs, where the operating system (written in Scratch) comes with "user-space" applications such as word-processors, etc. These applications have a "save" feature (e.g. one can save a document created in the word-processor), but in order for the saving functionality to work, the user has to run the Scratch project off the local authoring environment, and save the project after closing ("shutting down") the operating system session (figure 4-4). The act of saving the project creates a snapshot of the contents of the various Scratch data-structures, and the next time the project is loaded, and the operating system is booted up, the data-structures retain their previous state, and the user gets back his saved documents.



Reflex Tester

This is a reflex testing project. Click on the start button below to start the test. After a few seconds, the screen will change color. When that happens, click on the stop button. Repeat this 5 times. At the end, you'll be shown your score (average reaction time) and some questions to answer. Enter the answers and your score in a comment. Thank you very much for helping me with my science project!

(a)



[redacted] **8 months, 2 weeks ago**

I dont play any team sports. But I run and ski. Only a few hours a week. My score 286

[redacted] **9 months, 1 week ago**

1. Yes 2. Softball, Basketball, volleyball 3. 4-8 4 109.2

[redacted] **9 months, 4 weeks ago**

I used to swim for 4 hours a week. I got a score of 336.8

[redacted] **10 months ago**

I played volleyball and I practiced 2 hours a week. My score was 675.8

[redacted] **11 months ago**

used to play bace ball and mine was... 640.8

(b)

Figure 4-5: *A survey project in Scratch and the website comments being repurposed to store responses*

Scratch surveys

The workaround used for Scratch Operating Systems does not work for the current version of the Scratch website as the Java applet that runs the program online (in the browser) does not have a save functionality. So, in the online context, community members have often resorted to repurposing existing features of the website to store data. For example, to do a survey of the reaction time of his peers, a member of the community created and shared a program that would measure and display the time that a user of the program took to react to the color of the screen changing. After sharing the program, he asked members of the community to report their response times (and some additional data) through the comment system of the website (figure 4-5). After a pre-determined period, he manually collated the responses together to create a report on the average reaction time of community members (the report was in the form of another Scratch project). In this example, the programmer is already exploring some of the topics that I have listed in chapter 3 (viz. building data, representing data, etc.), but with a significant amount of effort and thought put into the act of appropriating (or "hacking") existing website features. In my design process, one of my goals was to enable these practices without the need to resort to these work-arounds.

Collaborations over the Scratch "mesh"

Scratch 1.4 has an experimental "mesh" feature that allows different Scratch programs on networked machines to communicate with each other. The communication happens via sharing of variables (variables from one program show up as a read-only sensor blocks in another program) and through sharing of "broadcasts" messages. However, setting up a mesh system is complex, requiring knowledge about the IP address of different machines, and even if set up correctly, the communication often fails to happen over firewalls and routers. For this reason, the "mesh" feature in the current release of Scratch is obfuscated, and can only be enabled through a special, hidden UI gesture. Additionally, the feature fails to work with the online Java player applet.

However, in spite of the mesh being normally inaccessible, enthusiastic members of the community have used it to create a multitude of projects, starting from chat rooms to complex multiplayer games. Most of these members are definitely advanced and dedicated users of Scratch, and using the mesh requires skills and expertise that is beyond that of the beginning, or even average Scratch user.

4.3 Related Systems and Prior Art

Apart from surveying community practices, for design inspirations, I also looked at systems which deal with persistent and/or shared data, both within the area of programming language theory and the space of visual programming languages for novices.

Programming and Data Persistence

A significant amount of research has been done around the space of programming with persistent data-structures, even if we look outside of the area of Relational Database Management Systems

(RDBMS) or Object-Relationship-Mapping (ORM) like techniques. I describe two relevant systems below:

PS-algol

Atkinson et al. [31] designed PS-algol, a language based on S-algol, to natively support data persistence. In PS-algol, the abstraction of data persistence was implemented in an interesting way—instead of letting the programmer determine the persistence of a data-structure, it was determined automatically. However, the data-store was defined manually, and the programmer had to explicitly choose a local on-disk data-store at the start of the program code for all PS-algol programs. Additionally, parallel instances of PS-algol programs were not allowed to write to the data-store at the same time, though parallel reads were possible.

Partial and full persistence

Driscoll et al. [32] had a different take on persistence where they defined a data-structure as persistent if "it supports access to multiple versions". They differentiated between two categories of persistence, partial and full. In partial persistence, only the last value of a data-structure could be modified, though all other versions can be read, while in full persistence, all versions of a data-structure could be read and modified.

Novice Programmers and Data

Both the system and concept described above were research projects that did not necessarily take into account the needs of novice programmers. Systems that allow novice programmers to explore data, especially online data have started to show up only recently, and I present two examples below—TinyWebDB and NetScratch:

TinyWebDB in App Inventor

App Inventor [33] is a visual block based language for developing mobile apps for Google Android based devices. The App Inventor programming toolkit has a component called TinyWebDB that allows programmers to store key-value pairs in an online web-service (figure 4-6). While this allows for persistent data (as well as sharing of data), the programmer

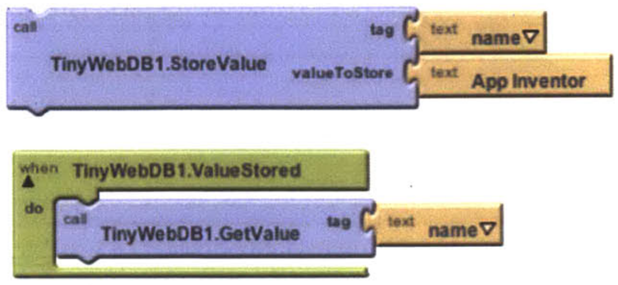


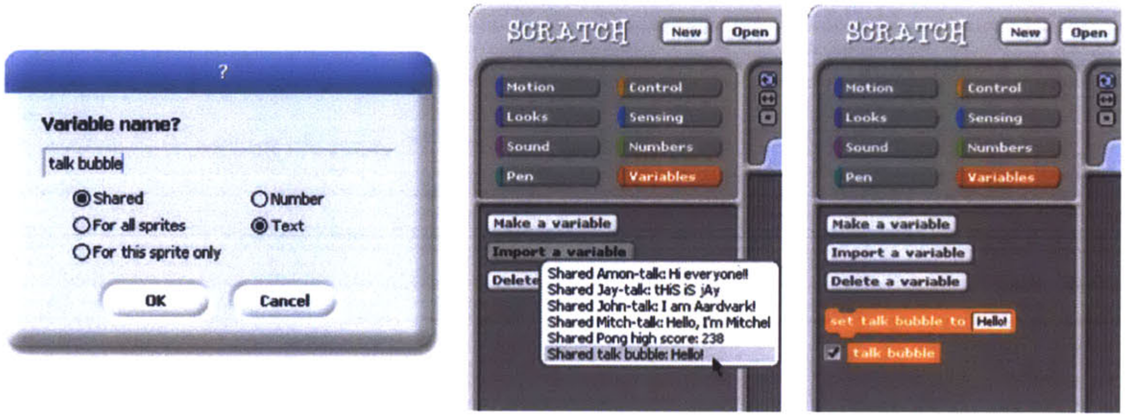
Figure 4-6: *Getting and setting a value with App Inventor's TinyWebDB*

has to explicitly define the connection parameters, and use a key-based lookup mechanism to make use of the system. Moreover, although a sample web-service exists for use with TinyWebDB (both as a running instance, as well as in source code form), serious users of TinyWebDB are expected to develop and deploy their own web-services.

Shariables in NetScratch

Tamara Stern [34] designed an online, shared variable system for Scratch called "shariables" as a part of a larger Scratch-derived project called NetScratch. Shariables were built on the standard Scratch variable system, with the additional property of being stored in a central server (figure 4-7). Shariables were associated with a given Scratch user, and since they could be used in multiple projects, they allowed for communication between different projects. One of the trickier aspects of using shariables was managing permissions, specifically determining who can read from and write to a shariable. According to Stern, feedback from the initial user tests of the system highlighted the need for a better access-control system:

> "As anticipated, the group wanted metadata associated with *shariables.* They wanted to know who created the *shariable* and who last updated it. This additional information became especially desired when the children "hijacked" (a term they used to describe someone mysteriously changing the value of a *shariable*) each other's *shariables.* They also wanted *shariables* to have read/write protections associated with them, assigning different users with different capabilities."



Variable name?
talk bubble

[✓ Shared] For all sprites For this sprite only

Number [✓ Text]

Shared Amon-talk: Hi everyone!!
Shared Jay-talk: tHiS iS jAy
Shared John-talk: I am Aardvark!
Shared Mitch-talk: Hello, I'm Mitchel
Shared Pong high score: 238
Shared talk bubble: Hello!

set talk bubble to Hello!
✓ talk bubble

Figure 4-7: *Creating and importing a shared variable (shariable) with NetScratch*

For my final design, I took lessons and insights from these systems and the community practices, and connected them to some of the computational and pedagogical inspirations that I have outlined in previous chapters. This design is described in the chapter that follows.

"Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away."

Antoine de Saint-Exupéry

5 Programming with Online Data

In this chapter, I describe the extensions that I built, starting with descriptions of the rationales for design choices in terms of goals and guidelines. I connect back these goals and guidelines to the inspirations described in the previous chapters. Later in the chapter, I also describe some of the social effects and implications of the design, along with possible ways of addressing those implications.

5.1 Design Goals

From the potential use-cases (Chapter 2), the computational practices around online data (Chapter 3) and the community practices (Chapter 4), a couple of affordances of Scratch 2.0 data-structures emerge as design goals—*data persistence* and *data shared-ness.*

Data Persistence

The property of *persistence* gives the programmer the ability to retrieve and reuse data that was created and subsequently stored during a previous execution instance of the program.

Data Shared-ness

While the property of *"shared-ness"* can be initially hard to understand and utilize for some (discussed in Chapter 7), it considerably expands the space of possible projects. Moreover, in the online context, being able to share data, either between parallel instances of the same program, and/or between different programs, is a somewhat "natural" or "expected" affordance. For my design, I explicitly chose the former mode of sharing (between instances), due to reasons that I elaborate later in this chapter.

5.2 Design Guidelines

One of my main goals in building the system was to make it generative, and hence all my design guidelines are linked to generativity in one way or the other. The core set of principles and guidelines for my design were:

- Simplicity
- Learnability
  - Reusing and connecting to existing metaphors
  - Supporting incremental learning

5.2.1 Simplicity

One of the simplest methods to deal with online data is to use a key-value pair system to store and lookup data. Apart from being used in App Inventor's TinyWebDB component, the key-value metaphor (albeit in a slightly more sophisticated form) is also used in very large scale for Amazon Simple Storage Service (S3) [35]. In the initial stages of my design, for a while, I considered a block based, online key-value data storage system with each storage instance being associated with a specific project, not unlike "buckets" in Amazon S3. However, as I tried to design the blocks, I realized that I was building a system that was a close parallel to the existing variable and list system in Scratch. The only major difference was the use of a key instead of a variable name or a list name. For the next iteration of my design, I decided to apply John Maeda's First Law of Simplicity, which states that *"the simplest way to achieve simplicity is through thoughtful reduction."*[36]. I performed my "thoughtful reduction" by collapsing the above two systems together, and ended up with a single, orthogonal property for Scratch 2.0 variables and lists. If this property is set, the data in the variable or list gets stored online. This formed my final design, and the data-structures with the above property set are named *Cloud* data-structures.

In my mind, the idea of *simplicity* relates to the idea of a "low floor", which again, is linked to generativity. Cloud data-structures are *structurally simple*—the only new feature that is added to the programming system is the ability to store data online (and retrieve it subsequently). There is absolutely *no* change whatsoever to the programming language grammar.

Apart from lowering the barrier to entry (low floor), this structural simplicity also leads to a more general space of applicability. As Resnick and Silverman put it, this leads to a "diversity of outcomes"[12]—the design ends up not being for specific activities such as making a user management system for Scratch 2.0 projects, or a score-keeping system for Scratch 2.0 projects, but for a more general and larger space which children can explore according to their own tastes and passions[1]. This again, is the idea of "wide walls", which links back to the concept of generativity.

[1]The flip-side of structural simplicity is, however, increased difficulty of beginners to explore the full range of functional possibilities, something which I briefly describe in the next section, and again, in Chapter 7.

5.2.2 Learnability

Apart from structural simplicity, the design of Cloud data-structures build upon already established metaphors, methods and functionality, something that I collectively term as *learnability*. The idea of learnability also connects to the "low floor" principle, and I try to ensure learnability through two main design principles, following the guidelines articulated by Mitchel Resnick in his Masters thesis [37].

Reusing and connecting to existing metaphors

In my final design, operations on "Cloud data-structures" are exactly the same as operations on standard Scratch 2.0 data-structures. The only difference lies in the process of declaring (or in Scratch terminology, creating) the data-structure instances, where an extra option needs to be selected through the user interface (figure 5-1). This means that young programmers who are already familiar with using standard Scratch 2.0 data-structures would be able to get started using Cloud data-structures right away.

However, it should be pointed out here that relying on a single metaphor has the danger of leading to a condition that was described by Alan Perlis as the "Turing tar-pit" [38], where he offers the following cautionary advice:

> "Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy."

As it emerges later in Chapter 7, there were certain project genres (such as multiplayer games) that could have been done more easily if a different metaphor was used, and there was also some initial confusion amongst the children using the system about the shared nature of the Cloud data-structures. Some of these difficulties and confusion can be traced back to the large functional space opened up by the design, with a comparatively simple structural change.

Supporting incremental learning

The previous section showed how the grammar for Cloud data-structures are exactly the same as the grammar for standard Scratch 2.0 variables and lists. In addition to this, the functionality of standard Scratch 2.0 data-structures is purely a subset of the functionality of Cloud data-structures. Thus, it is entirely possible for a Scratch 2.0 programmer to use a Cloud data-structure in place of a standard Scratch 2.0 data-structure, assuming the data-structure is initialized properly in the program code. So effectively, a programmer who does not know what a Cloud data-structure is, but tries to use it nevertheless would not see any difference in program behavior initially. However, once he learns about the extra functionality of Cloud data-structures (either through looking at other programs in the community, or through some explicit piece of information, such as a tutorial), he can quickly and easily modify his program to take advantage of the functionality.

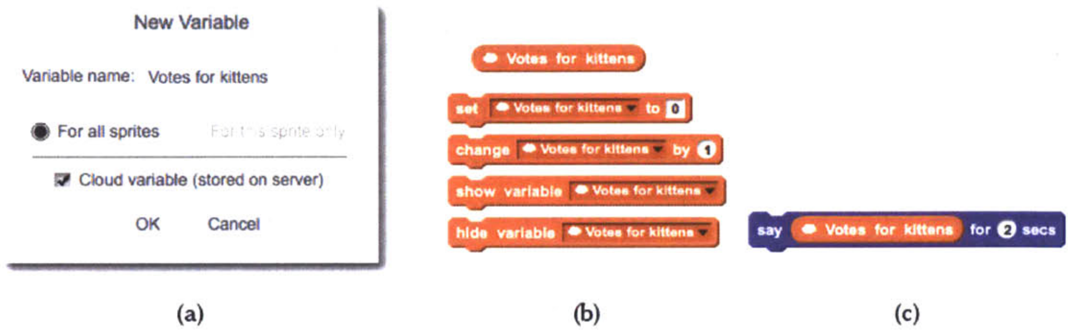
5.3 Design Description

While my design efforts were primarily intended for the *programmer* using Scratch 2.0, the implications of the design affect the interactions available for the *end user* of the program as well. In this section, I describe the system as the programmer sees it, as well as what the implications are for the end-user interaction of a program that uses Cloud data-structures. In addition, I also compare and contrast the design with those of existing systems, specifically NetScratch and App Inventor.

Interactions for the Programmer Using Scratch 2.0

Unlike traditional programming languages, Scratch 2.0 data-structure instances are created through UI gestures in the development environment, and not dynamically in the code. Cloud data-structures follow a similar interaction pattern, where a programmer explicitly creates a data-structure instance while selecting the option "*Cloud data-structure (stored on server)*" (figure 5-1a).

Once the data-structure instance is created, a "reporter" block (with a ☁ indicator icon) for it shows up in the development environment's palette, along with blocks for other operations on the data-structure (figure 5-1b). The "reporter" block can be embedded into other Scratch 2.0 blocks in the actual program code (figure 5-1c).


New Variable

Variable name: Votes for kittens

[✓ For all sprites] For this sprite only

[✓ Cloud variable (stored on server)]

OK
Cancel
Votes for kittens
set Votes for kittens to 0
change Votes for kittens by 1
show variable Votes for kittens
hide variable Votes for kittens
say Votes for kittens for 2 secs
(a)
(b)
(c)

Figure 5-1: *Programmer interactions for using Cloud variables. (The interaction-set for Cloud lists is similar)*

The operations for a Cloud Variable are:

- set
- get (as a "reporter" block)
- change by (increment or decrement, depending on the sign of the argument)

The operations for a Cloud list are:

- add item (append)
- delete item
- insert item
- replace item
- item at a given index (as a "reporter block")
- length
- membership test (as a boolean "reporter block")

As mentioned earlier, in both cases, all the operations are exactly the same as normal Scratch 2.0 data-structures.

Interactions for the End User

Initially, the interactions for the end user (i.e. the person who runs a project created by someone else) appear to be the same as with any other Scratch 2.0 project. However, the Scratch 2.0 website also allows for remixing of projects, which means *(a)* an user can look at (and tinker with) the code of any project following which, *(b)* she can click on the "remix" button to reuse the code in her own project. With these possibilities, we can consider the following scenario:

> *A Scratch community member develops a game with a high-score list that is implemented with a Cloud list. A number of community members find the game, play it, and soon, there is a significant number of entries in the high-score list. One particular member, however, decides to "cheat" and adds his name to the high-score list without playing the game. To do this, he clicks on the "See Inside" button beside the project to open up the code-editor, drags out the block to append items to a list, and adds his name to the high-score Cloud list by clicking on the block (Scratch allows for the execution of any block by clicking).*

To deal with this type of situations, the Scratch 2.0 code editor goes into a special sandbox mode when anyone other than the program's creator starts to look at a program's code. In the sandboxed mode, update operations on Cloud data-structures are recorded, but not committed to the online data-store (a warning dialog pops up if someone tries to change the value of a Cloud data-structure). Only when the remix button is clicked, the transfer of ownership takes place, the project gets a new ID number, and all changes to the Cloud data-structures get committed to the online data-store, associated with the new project ID. So, with these safeguards in place, in the example above, the user who tries to add his name to the high-score list without playing the game would not be able to do so unless he remixes the project.

Contrasting with Existing Systems

The design of Cloud data-structures differs from those of existing systems like NetScratch and App Inventor in subtle, but significant ways. In this subsection, I highlight the key differences between choices that I made in my design, and those made by the developers of NetScratch and App Inventor.

NetScratch

In Stern's thesis, one of the conclusions about shariables in NetScratch was the need for better manageability. Manageability can be decomposed into two main problem areas—scalability and access control. Shariables were associated with individual users on the Scratch website, and as the number of users grew (the website has more than a million registered users as of July 2012), it would have become increasingly difficult to search for relevant shariables. Also, with Stern's implementation, any user could write to, or read from any shariable, something that could be easily abused, or as users testing the system called it, "hijacked". To counter this, an access control system could be used, but I felt that making a easy-to-understand access control system would be difficult.

The approach I took to ensure manageability was to associate Cloud data-structures with projects, rather than users. Each instance of a Cloud data-structure "belongs" to a project, and is accessible from only within the project. This leads to the trade-off that there is no way to use Cloud data-structures for inter-project communications. However, on the other hand, one of the consequences of this design is the fact that the question of searching for Cloud data-structures does not exist any more—a given Cloud list or variable is relevant and available only in the context of a single project. As for access control, contents of Cloud data-structures are accessible to users running the project. However, the exact details of the access (the specific access control rules) are determined by the programmer of the project—read access, as well as write access is mediated through the code. If the programmer wants her users to be able to add to a given Cloud list in her project, she has to write the code to allow users to do that, and if she wants read-only access, that also, has to be enabled with code.

Figure 5-2 illustrates the design space of data-structures in Scratch, and shows the relative points in the space that are occupied by standard Scratch variables, shariables and Cloud variables.

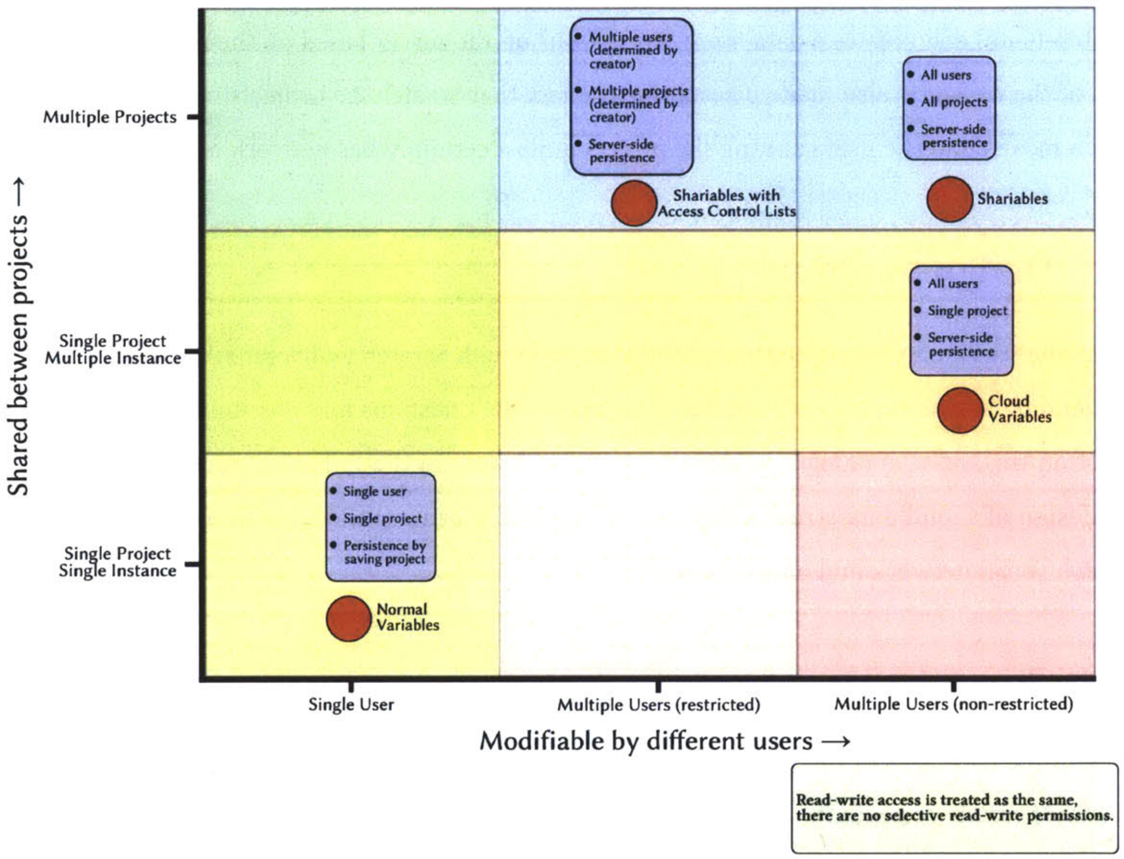

Shared between projects →
Multiple Projects
Single Project Multiple Instance
Single Project Single Instance
• Multiple users (determined by creator)
• Multiple projects (determined by creator)
• Server-side persistence
Shariables with Access Control Lists
• All users
• All projects
• Server-side persistence
Shariables
• All users
• Single project
• Server-side persistence
Cloud Variables
• Single user
• Single project
• Persistence by saving project
Normal Variables
Single User
Multiple Users (restricted)
Multiple Users (non-restricted)
Modifiable by different users →
Read-write access is treated as the same, there are no selective read-write permissions.

Figure 5-2: *Design space for online data blocks—showing Shariables and Cloud data-structures.*

App Inventor

App Inventor's TinyWebDB is a flexible system to retrieve and store online data using a key-value mechanism. The TinyWebDB component can connect to a web-service specified by the programmer, and multiple App Inventor projects (apps) can use a shared web-service to communicate among each other. However, this web-service has to be programmed and deployed by the programmer, which increases the barrier to entry for using TinyWebDB considerably[2].

In my final design, the Cloud data-structure system does not require any explicit step or configuration to connect to the online data-store. Since Cloud data-structures are associated with a project, the Scratch 2.0 interpreter automatically establishes a connection to the server (maintained by the Scratch-team) and creates a data-store "collection" on the server based on the project ID. This aspect of the design is also made possible by the fact that Scratch 2.0 projects run on the browser, which means that the user running the project almost certainly has network connectivity.

5.4 Design Implications

Given the social nature and context of programming with Scratch and Scratch 2.0 [39], the design of Cloud data-structures raises a number of interesting questions and possibilities in terms of its effect on the social dynamics on the site and vice-versa. In the following sections, I show how the design of Cloud data-structures started to affect the *privacy* and *moderation* dynamics of the Scratch social network and online community.

5.4.1 Privacy

A lot of the use-cases of the Cloud data-structure system require the ability to programmatically identify Scratch community members who are accessing a project. For example, in a survey project, unless a community member can be identified from within the project's code, there is a risk of someone skewing the results by voting repeatedly for a particular choice. The initial survey projects used the "ask" block in Scratch 2.0 to identify a user, but that approach was entirely dependent on the user typing in the username correctly and/or truthfully. One possible solution to this problem was to provide a reporter block for the username of the user running the project, and in the initial iterations of the implementation I included a "Username" block with this functionality. However, this gave rise to a number of privacy concerns. With the username block,

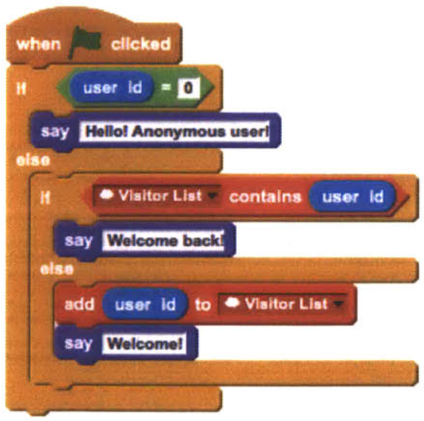
[2]The App Inventor tutorial includes a sample template web-application that can be used in conjunction with TinyWebDB, but even with this template code, absolute new users will possibly feel intimidated by the prospect of deploying a web-service all by themselves.

one could easily record actions of users in an identifiable manner, and even write special code for specific users (e.g. make a sprite say inappropriate messages for a certain user). Ultimately, as a workaround for these issues, I implemented an *User ID* block as a potential middle-ground solution.

The User ID block

With the *User ID* reporter block (figure 5-3), the value returned by the reporter is unique for a given username and project combination. Internally, the block keeps track of the users visiting the project, and assigns each non-anonymous user the next highest integer. Thus, the first logged-in user to visit a project always has the user ID of 1 for that project (even for subsequent visits), the second visitor always has the ID 2, and so on. Anonymous or non-logged in visitors always have the ID of 0.

An initial concern was whether the User ID block was too "black-boxy", and whether the fact that it reported/returned an unique number for each user-project combination was expressed clearly enough. However, from initial user-feedback, it seems that community members understood the functionality of the block well enough to start experimenting with it and using them in their own projects, without any implicit instruction or support from us.



```
when ⚑ clicked
if user id = 0
  say Hello! Anonymous user!
else
  if Visitor List contains user id
    say Welcome back!
  else
    add user id to Visitor List
    say Welcome!
```

Figure 5-3: *User ID block in action—the program keeps track of visitors in a Cloud list, and sprite says the appropriate welcome message.*

5.4.2 Moderation

Another major question around the design and use of Cloud data-structures is around safety and moderation, and going down a bit deeper, in this case, there are two questions to address—moderating end users and moderating programmers.

Moderating end users

A comparatively simple project with Cloud data-structure is message-box or shout-box project, where people can chat asynchronously (a synchronous chat program is also quite easy to implement). There is a very real danger of someone typing in inappropriate messages in such a project, and it is difficult to moderate these messages, as a very large number of messages can be generated in little time[3]. As one of the early testers (who is an educator) commented:

> "I'm excited about cloud variables. I'm curious, however, how this may entangle with privacy issues. For example, what if my students use scratch to make a chat room or a facebook-alike? Will the content of such a system be subject to the same awesome community moderation that monitors scratch projects now?"

Currently the underlying infrastructure code keeps track of all operations on Cloud data-structures, along with the username of the person initiating the operation. Later, the logging mechanism may also incorporate an automatic flagging mechanism that would alert moderators if any incoming data contains potentially inappropriate content. There is also a proposal to restrict Cloud data-structures to only users who are logged into the Scratch 2.0 website, so as to minimize the chance of anonymous users adding inappropriate data. None of these approaches prevent inappropriate content, but rather, they are intended to help me and the larger team behind Scratch keep track of and understand the problem, and design appropriate responses.

[3]The Scratch community is designed to be a safe space for children of all ages, and has an active moderation system in place.

Moderating programmers

While the previous category of concern is more about *"malicious" users*, there is also a real danger of *"malicious" programmers* trying to take advantage of unsuspecting users. With persistence, it is easy to create projects that can misguide users to divulge personal information and store these information in Cloud data-structures. For example, it is trivial to create a project that claims to have encountered an "error condition" and presents a fake login dialog asking for the user's username and password (a Scratch 2.0 equivalent of "phishing").

For these sort of projects, as with phishing in the real world, perhaps the most effective solution is raising awareness in the community [40].

However, to conclude, with respect to both the questions outlined above, I should perhaps note that these dangers and potential drawbacks are inherent characteristics of any generative system. As Zittrain points out [10],

> "Generative systems are threatened by their mainstream success because new participants misunderstand or flout the ethos that makes the systems function well, and those not involved with the system find their legally protected interests challenged by it. Generative systems are not inherently self-sustaining when confronted with these challenges."

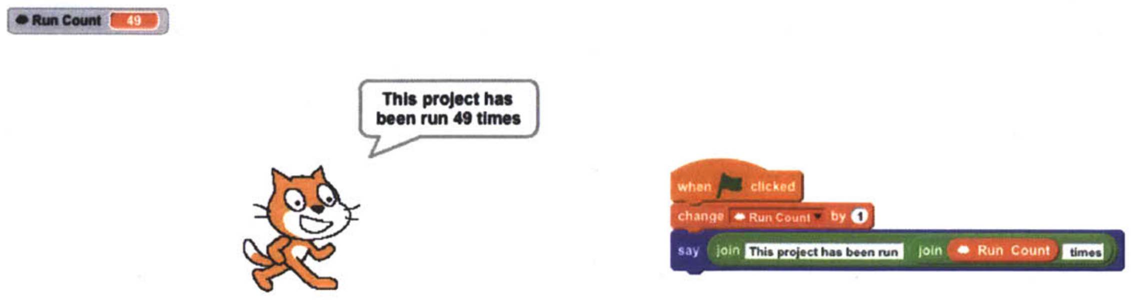
With the design outlined in this chapter, in a relatively short period of time, and with a very small user community, a comparatively wide variety of projects were created, some of which are described in detail in the next chapter.

"Adults worry a lot these days. Especially, they worry about how to make other people learn more about computers. They want to make us all 'computer-literate.' Literacy means both reading and writing, but most books and courses about computers only tell you about writing programs. Worse, they only tell about commands and instructions and programming-language grammar rules. They hardly ever give examples. But real languages are more than words and grammar rules. There's also literature—what people use the language for. No one ever learns a language from being told its grammar rules. We always start with stories about things that interest us."

Marvin Minsky [41]

6 Example Projects

This chapter provides a sampling of projects that were created with Cloud data-structures. The list is a mix of projects that I created as examples, as well as ones that were created by participants in the workshops that I ran, and by alpha testers of the Scratch 2.0 prototype website. The primary purpose of this list is to provide a sense of diversity of the projects that were created using Cloud data-structures in a comparatively short period of time by a small group of Scratch programmers.



```
when [flag] clicked
change Run Count by 1
say join This project has been run join Run Count times
```

(a) (b)

Figure 6-1: *Project run counter and corresponding code*

6.1 Project Run Count

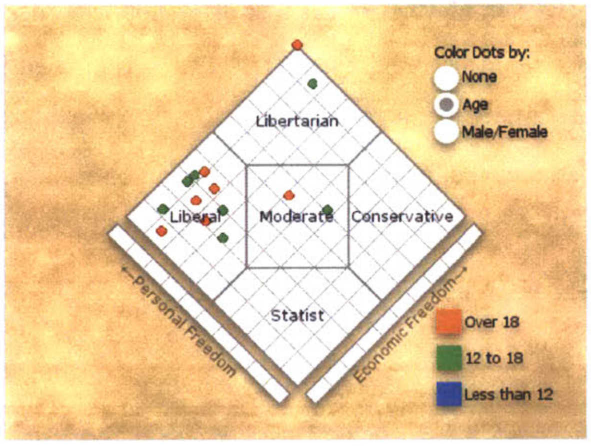
As an example to introduce Cloud data-structures, I came up with a project that keeps track of how many times it has been run.

Execution of a Scratch project is typically started when the user clicks on a *green flag* icon on the Scratch "player". The click triggers an event-handler block (called a "hat" block in Scratch terminology), and in this particular project, the green flag hat block had a block-stack attached to it that incremented a Cloud variable called Run Count by 1 (figure 6-1 (b)). As Cloud variables are persistent, with this event-handler block-stack, Run Count's value was always equal to the number of times the green-flag has been clicked, or in other words, the number of times the project has been run. Apart from incrementing the value of Run Counter, I also included a block that made the cat on the Scratch stage say the number of times the project has run (figure 6-1 (a)).

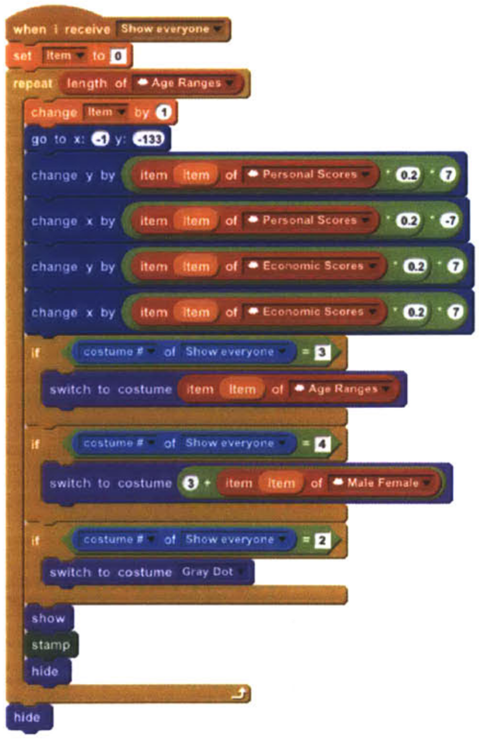
6.2 Political Orientation Survey

Political Orientation Survey was designed to be a survey of the political beliefs (liberal, conservative, libertarian, etc.) of the members of the Scratch community. Though workshop participants and testers of the Scratch 2.0 prototype created a number of survey projects, I chose this particular one for its strong design focus.

First time viewers of the project (determined through the User ID block) were asked to choose their age-group and gender, and then were provided with a set of "Yes/No/Maybe" options for a number of social and economic policy questions. At the end of the questionnaire, the user was placed (anonymously) on a "political map"—a two dimensional scatterplot graph with the axes representing scales for economic and personal freedom. The map also showed the positions of previous respondents (stored in Cloud lists), so that the user could get a sense of the opinion of others and compare her own overall position with those of previous respondents. The scatterplot also provided options to color-code the results according to the demographic parameters mentioned earlier—age group and gender (figure 6-2). When the user revisited the project, she was taken directly to the results map, again using the User ID block.



(a)



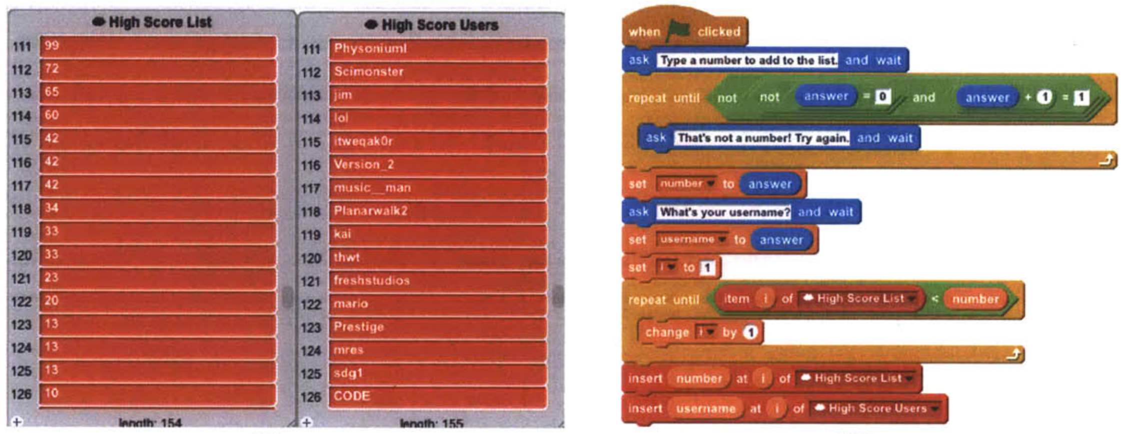
```
when I receive [Show everyone]
set [Item] to [0]
repeat (length of [Age Ranges])
    change [Item] by (1)
    go to x: (-1) y: (-133)
    change y by ((item (Item) of [Personal Scores]) * (0.2) * (7))
    change x by ((item (Item) of [Personal Scores]) * (0.2) * (-7))
    change y by ((item (Item) of [Economic Scores]) * (0.2) * (7))
    change x by ((item (Item) of [Economic Scores]) * (0.2) * (7))
    if <([costume #] of [Show everyone]) = [3]>
        switch to costume (item (Item) of [Age Ranges])
    if <([costume #] of [Show everyone]) = [4]>
        switch to costume ((3) + (item (Item) of [Male Female]))
    if <([costume #] of [Show everyone]) = [2]>
        switch to costume [Gray Dot]
    show
    stamp
    hide
hide
```

(b)

Figure 6-2: *Political orientation survey project and parts of the corresponding code*

6.3 High Score List

The high-score list project was a collaboration between a couple of the testers of the Scratch 2.0 prototype website.



	High Score List		High Score Users
111	99	111	Physoniuml
112	72	112	Scimonster
113	65	113	jim
114	60	114	lol
115	42	115	itweqak0r
116	42	116	Version_2
117	42	117	music__man
118	34	118	Planarwalk2
119	33	119	kai
120	33	120	thwt
121	23	121	freshstudios
122	20	122	mario
123	13	123	Prestige
124	13	124	mres
125	13	125	sdg1
126	10	126	CODE
	length: 154		length: 155

```
when clicked
ask [Type a number to add to the list.] and wait
repeat until <not <<not <(answer) = [0]> and <((answer) + (1)) = [1]>>>
    ask [That's not a number! Try again.] and wait
end
set [number] to (answer)
ask [What's your username?] and wait
set [username] to (answer)
set [i] to [1]
repeat until <(item (i) of [High Score List]) < (number)>
    change [i] by (1)
end
insert (number) at (i) of [High Score List]
insert (username) at (i) of [High Score Users]
```

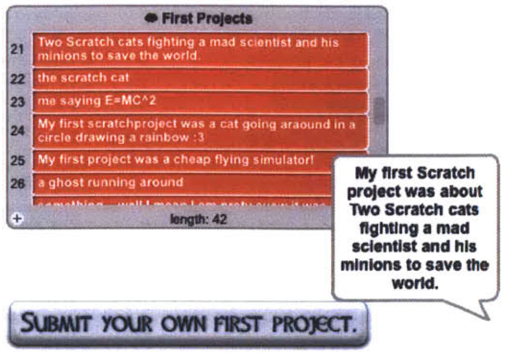
(a) (b)

Figure 6-3: *High-score list project with corresponding code*

The first project of the collaboration was a single sorted Cloud list of numerical scores. This was then remixed (figure 6-3) by another tester to create a high-score list that also supported user-names. Internally, the second project used two Cloud lists and ensured that they remained in sync (i.e. the index of the user in the user list was the same as the user's score in the score list). The high-score list project was one of the case-studies that prompted the discussion on the user-name/user-id block (described in the previous chapter), as users of this project had to manually enter their names, something that was potentially error-prone and open to manipulation.

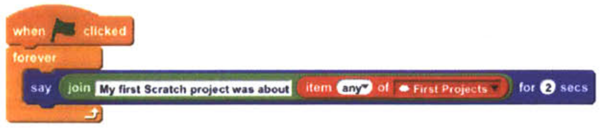
6.4 My First Scratch Project

I created the "My First Scratch Project" as a simple example demonstrating crowdsourcing with Cloud data-structures. The aim of the project was to crowdsource a collection of "subject of my first Scratch project" from members of the Scratch online community. The project used a Cloud list to store this collection.



First Projects
21 Two Scratch cats fighting a mad scientist and his minions to save the world.
22 the scratch cat
23 me saying E=MC^2
24 My first scratchproject was a cat going araound in a circle drawing a rainbow :3
25 My first project was a cheap flying simulator!
26 a ghost running around
length: 42
My first Scratch project was about Two Scratch cats fighting a mad scientist and his minions to save the world.
Submit your own first project.

(a)



```
when [flag] clicked
forever
    say (join [My first Scratch project was about] (item (any) of [First Projects])) for (2) secs
```

(b)

Figure 6-4: *"My first Scratch project" crowdsourcing project, with parts of the corresponding code*

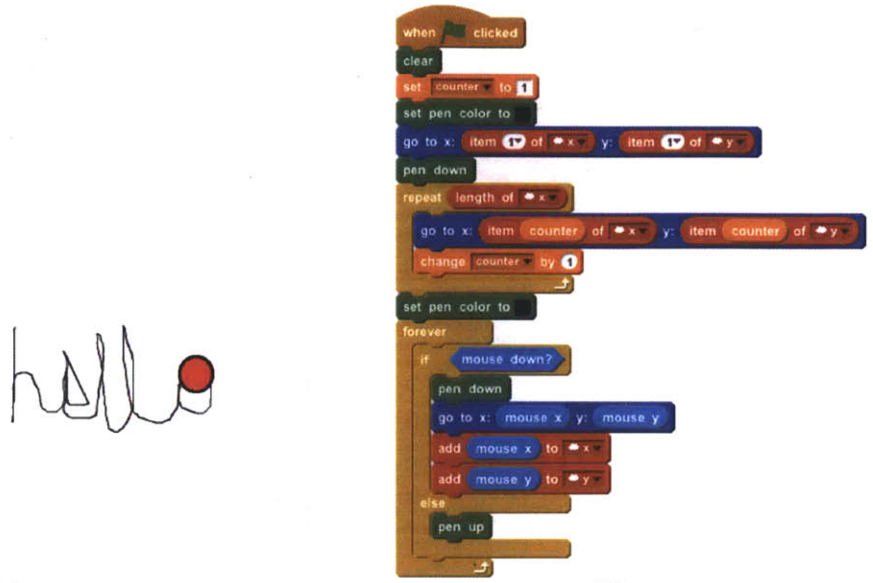
During execution, the project cycled through the list of subjects of first projects in a random order (figure 6-4), with each subject being shown for two seconds. Also, the user viewing the project could add his own first project to the list, and it would then be shown to subsequent viewers.

6.5 Collaborative Canvas

Collaborative canvas was an example to show how user interactions could be recorded and shared asynchronously using Cloud data-structures.

In this project, I used two Cloud lists to track and record the movements of the mouse over the Scratch stage, which was being used as a canvas (the mouse cursor being the pencil/drawing tool). When another user visited the project, the Cloud list pair was used to "play-back" the interactions of previous users on the canvas (stage), and the user could then add to the set of interactions to build a collaborative drawing (figure 6-5). This was also a project that showcased how Cloud data-structures could be used for asynchronous collaboration, where multiple users could contribute to a joint project (without even remixing).

The example project was remixed by a member of the tester community, with new features as well as optimizations to the user-interaction recording code.



```
when [flag] clicked
clear
set counter to 1
set pen color to
go to x: (item 1 of x) y: (item 1 of y)
pen down
repeat (length of x)
    go to x: (item (counter) of x) y: (item (counter) of y)
    change counter by 1
set pen color to
forever
    if <mouse down?>
        pen down
        go to x: (mouse x) y: (mouse y)
        add (mouse x) to x
        add (mouse y) to y
    else
        pen up
```

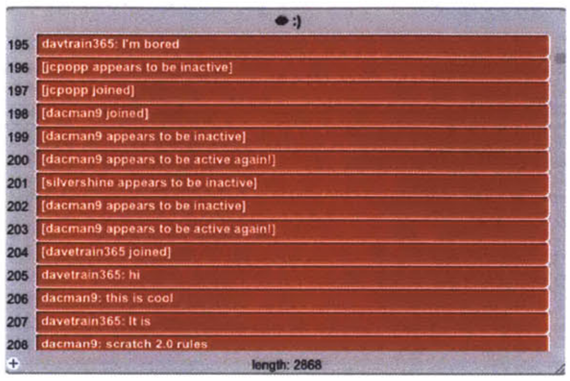
(a) (b)

Figure 6-5: *Collaborative canvas project with the corresponding code*

6.6 Chat with Idle Detection

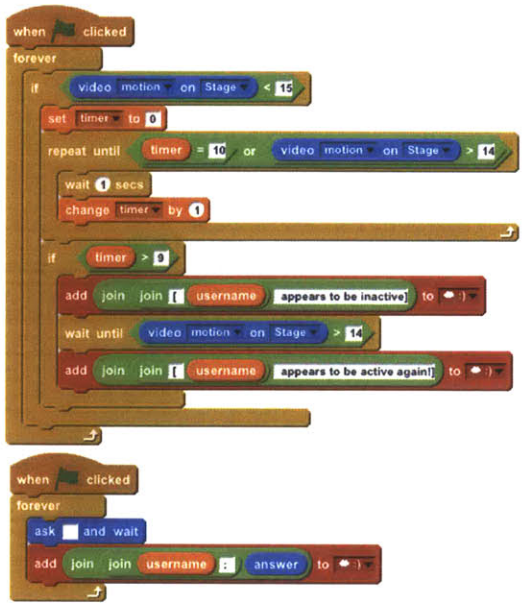
The Chat project was another example that I created to demonstrate the shared-ness property of Cloud data-structures. In this project, messages from users were appended to a Cloud list. Since the Cloud list is shared, updates to the list were propagated to parallel instances of the program immediately, allowing for real-time, synchronous chat. Additionally, due to persistence, anyone who joined the "chat" program at a later stage got all the previous messages that had been exchanged. The program itself consisted of an infinite ("forever") loop that prompted for user input, waited, and on getting the input, appended it to a Cloud list. The display of the messages was handled by a "list watcher" – a feature in Scratch that lets anyone running a program watch the contents of a list data-structure in real-time.

The example I created was remixed by a tester to add an extra "idle detection" feature, which used the new webcam-based motion detection block in Scratch 2.0 to mark a chat participant as inactive if no motion was detected in a given amount of time (figure 6-6). This remix is an illustrative example of types of projects where Cloud data-structures are not necessarily in the center, but



```
● :)
195 davtrain365: I'm bored
196 [jcpopp appears to be inactive]
197 [jcpopp joined]
198 [dacman9 joined]
199 [dacman9 appears to be inactive]
200 [dacman9 appears to be active again!]
201 [silvershine appears to be inactive]
202 [dacman9 appears to be inactive]
203 [dacman9 appears to be active again!]
204 [davetrain365 joined]
205 davetrain365: hi
206 dacman9: this is cool
207 davetrain365: It is
208 dacman9: scratch 2.0 rules
length: 2868
```

(a)



```
when clicked
forever
    if video motion on Stage < 15
        set timer to 0
        repeat until timer = 10 or video motion on Stage > 14
            wait 1 secs
            change timer by 1
        if timer > 9
            add join join [ username appears to be inactive] to ● :)
            wait until video motion on Stage > 14
            add join join [ username appears to be active again!] to ● :)

when clicked
forever
    ask and wait
    add join join username : answer to ● :)
```

(b)

Figure 6-6: *Chat project with idle-user detection and it's corresponding code*

enable children to think about larger and broader ideas, such as in this case, interesting forms of human-computer interaction.

These projects form a sampling of the wide variety of creative possibilities that Cloud data-structures enable. In the next chapter, I revisit some of these projects, especially the ones that were created or remixed by Scratch programmers, looking in depth at the learning outcomes that emerge from them.

"Grown-ups never understand anything by themselves, and it is tiresome for children to be always and forever explaining things to them."

The Little Prince

7 Studies with Children

In this chapter, I describe a number of studies of the Cloud data-structure system with children in various settings and contexts and look at the learning outcomes, as well as confusions and misconceptions that arise from the system and its design.

7.1 Methodology

To understand the efficacy and implications of my design, I conducted both formal and informal studies with children in a number of settings, and gathered feedback through surveys, face-to-face conversations, and discussions on feedback forms. A listing of the contexts that I used for my studies follows:

Workshop I

The first study of Cloud data-structures was carried out in a local middle school Scratch club, with a couple of one-and-half hour workshops spread out over two days. Most of the participants were veterans of the Scratch club, with intermediate to advanced level experience with Scratch. There were in total 8 of participants, with 3 girls among them. 3 participants were in high-school, while the rest were middle school students.

Workshop II

The second workshop was of shorter duration (45 minutes), but with a larger number of participants, and more unstructured. This particular workshop took place as a part of a larger Scratch event at MIT, and was open to anyone who wished to join.

Scratch-2.0 Prototype Testers

A significant amount of feedback also came from the testers of the Scratch 2.0 prototype. These testers were mostly active members of the wider Scratch community, as well as educators and researchers, and most of them had advanced level experience with Scratch. Though the entire tester group had 54 members, I focused mostly on the younger members of the group.

7.2 Findings

With my focus on generativity and learning, I was interested not only in the diversity of projects that were made using the system, but also in what the users of the system learned as a result. The previous chapter includes a sampling of the creative outcomes from the user studies (along with a set of sample projects that I designed to be "seeds for inspiration"). This section consists of descriptions of my observations related to the learning outcomes, along with a few selected insights that I gathered from looking at the projects, from the conversations, and from the feedback that I got through the surveys. Additionally, I was also interested in children's models of the internal workings of the Cloud data-structure system, and in the final part of this section, I describe the mental models for Cloud data-structures, as described by participants from my first workshop.

7.2.1 Learning Outcomes

In terms of learning outcomes, through the projects, participants of the workshops and the testers demonstrated understanding of a number of ideas, concepts, practices and perspectives that are related to programming with online data. In trying to enumerate the learning outcomes, I analyze three projects, and connect back the results of the analysis to the topics that I list in chapter 3 (building data, processing data, representing data and organizing data):

High score list with usernames

In the first iteration of the high-score list project (described in section 6.3), scores were inserted in order into the Cloud list, thus ensuring that the list was always kept sorted. This project was created by one of the testers of the Scratch 2.0 prototype, and algorithm 1[1] describes how it worked.

This project was remixed by another tester, who added support for "negatives, decimals, and usernames." He used two Cloud lists to achieve this, as shown in algorithm 2.

Algorithm 1: Algorithm for maintaining a sorted high-score list

```
validate the score as a non negative integer;
$i \leftarrow 1$;
while $HighScoreList[i] < score$ do
    // Increment $i$ by 1 for each iteration
    $i \leftarrow i + 1$;
end
Insert $score$ at position $i$ of $HighScoreList$;
```

Algorithm 2: Modified algorithm for maintaining a sorted high-score list

```
validate the score as a number;
$i \leftarrow 1$;
while $HighScoreList[i] < score$ do
    // Increment $i$ by 1 for each iteration
    $i \leftarrow i + 1$;
end
Insert $score$ at position $i$ of $HighScoreList$;
Insert $username$ at position $i$ of $UserList$;
```

[1] I use algorithms in this chapter, instead of screenshots of Scratch blocks, primarily in order to make the listings concise and more readable. However, for all the algorithms, I try to remain as close as the "spirit" of the original program as possible.

This remix was a simple, but effective improvement to the original algorithm where the remixer ensured that the index of the user and his score in the corresponding lists were always the same. Taken together, the two projects demonstrated understanding of building data (*schema design*, as well as *choice of data-structure*), as well as processing data (*sorting*).

Collaborative canvas

I had included a relatively simple collaborative canvas project as a sample (described in section 6.5). Users of the project would draw lines on the canvas by clicking and dragging mouse, and the coordinates of the points on the lines would get stored in two Cloud list, one for the x-coordinates, and the other for the y-coordinates (algorithm 3).

Algorithm 3: Initial algorithm for collaborative canvas

```
ClearCanvas;
// Draw the previous state of the canvas
PenDown;
$i \leftarrow 1$;
for length of xList do
    MoveToPosition($xList[i]$, $yList[i]$);
    $i \leftarrow i + 1$;
end
PenUp;
// Draw following the user's mouse position & record
repeat
    if mouseIsDown? then
        PenDown;
        MoveToPosition ($mouseX$, $mouseY$);
        Append $mouseX$ to $xList$;
        Append $mouseY$ to $yList$;
    else
        PenUp;
    end
until forever;
```

A member of the prototype testers remixed the project with an optimized approach (algorithm 4), with the following note:

"I remixed sdg1's project so more than one line can be used. I also optimized the script so if you are drawing very slowly and do not move your mouse very far, it will not add as many items to the list—so people don't have to wait a long time while they watch the cloud drawing."

Algorithm 4: Optimized algorithm for collaborative canvas (slightly simplified)

```
ClearCanvas;
// Draw the previous state of the canvas
MoveToPosition($xList[1]$, $yList[1]$);
$i \leftarrow 1$;
PenDown;
for length of xList do
    if $xList[i]$ == '–' then
        PenUp;
    else
        PenDown;
        MoveToPosition($xList[i]$, $yList[i]$)
    end
    $i \leftarrow i + 1$;
end
PenUp;
// Draw following the user's mouse position & record
repeat
    if mouseIsDown? then
        Append '–' to xList;
        Append '–' to yList;
        MoveToPosition (mouseX, mouseY);
        Append mouseX to xList;
        Append mouseY to yList;
        PenDown;
        repeat
            MoveToPosition(mouseX, mouseY);
            if Distance (currentPosition, previousPosition) > 5 then
                Append mouseX to xList;
                Append mouseY to yList;
            end
        until !mouseIsDown;
    end
until forever;
```

In this new algorithm, the changes were significant, and there were a number of optimizations (e.g. decrease in the number of recorded data-points, based on distance between two successive mouse positions) and features (e.g. ability to draw multiple lines). The changes demonstrated understanding of building data in the form of collecting data via *logging*, optimizing logging and using *markers* (–) to annotate the information in the logs.

Also, as a follow-up, after a number of other members of the community had used the project, the young programmer who created the optimized remix had further reflections to share:

> "Now that I see a lot of people using this, there's certainly some things to improve. Firstly, after a lot of people have used this, the screen gets quite cluttered. There's a couple solutions. One, is to have a clear button (clears the list and all of the pen marks). Another is to limit each list to something like 500 items, and if it exceeds that, then it will start deleting values from the beginning of the list. And of course we could add lines, colors, etc. Lots of potential! :)"

Political map survey

In the Political Map Survey (described in section 6.2), there was a strong focus on the design for representing data of the responses collected through the survey. Not only was a *scatter-plot* used to visualize the collected data, but as an extra, color-codes were used to show different properties of points on the scatter-plot (eg: gender, age-range, etc.). This demonstrated understanding of the important idea that properties such as color can be used to represent dimensions when spatial dimensions in a visual representation are exhausted.

7.2.2 Confusions and Misconceptions

Despite the variety of projects, and understanding of a variety of topics, there were a few difficulties, confusion and misconceptions around the Cloud data-structure system. I describe two of the most important difficulties that the young programmers faced below:

Understanding of shared-ness

During the workshops, a number of participants came up to me and asked if the data in Cloud variables and lists were accessible to other "programs", and if it were the case, whether the updates happened automatically. There may be two explanations for the first question—*(a)* the participants wanted data-structures that could be accessed from other projects (which I specifically did not enable as a design choice), or *(b)* the participants did not have a term for "program instance", or "process", and used the term "program" instead.

During my design, I thought it would be intuitive or natural to have the data-updates be propagated or "pushed" between parallel instances. However, the system does not provide any explicit feedback about that happening, and from the questions, I realized that the feature became apparent to the participants of the workshop only after I told them about it.

Another possible sign of shared-ness not being immediately obvious (and maybe, even difficult to grasp) is the lack of original projects that utilized shared-ness. While there were a number of remixes of the chat example that I provided (including the one I describe in section 6.6), there were no other completed projects that used the parallel-instance sharing. A number of the participants and testers talked excitedly about making multiplayer games with Cloud data-structures that would take advantage of the shared-ness, but none of the efforts to create such games bore fruit during the workshops or during the alpha-testing period.

Structural vs. functional understanding

The structural model of Cloud data-structures is simple, and is consistent across contexts. There are only a limited number possible operations on Cloud data-structures (e.g. set, append, etc), and all the operations are independent of specific use. The functional model, on the other hand, spreads over a vast number of uses, and is dependent on the required consequences or intent. For example, cloud variables can be used to keep count of votes or users, to keep track of project "state", to store user preferences, etc. Similarly, Cloud lists can be used for logging data, store user progress in games, keep high scores, etc. For this type of scenarios, in his essay "Models of Computation" [5], Andrea diSessa puts forward the argument that one needs a "repertoire" of functional models:

> "Functional models provide only a view of part of the system, and that only with respect to a non-universal frame of analysis. Obviously, one needs a repertoire of them; the notion of a single model is tenable structurally but not functionally."

Also, with a simple structure, and a broad set of functionalities, there is a need for greater "cleverness", which can cause additional problems for novices.

> "Specifically, the construction of a broad class of functionalities out of a tiny set of structural elements is almost bound to involve great cleverness. [...] Functional understanding is not well supported by a structural model. This side of a system will require specific tutoring[...]"

Thus, to enable learners to develop a wider "functional vocabulary", as diSessa puts it, my approach has been to provide as many "seed" sample projects as possible, which can be easily understood and appropriated by novices for their own projects.

7.2.3 Perspectives on Larger Computational Topics

Apart from concepts related to programming with online data, the studies also brought up certain valuable insights into the perspectives of the young programmers on certain larger, powerful ideas on computation—I highlight a couple of them below:

Perspectives on scale

Some of the children in the studies had a pretty accurate idea about the scalability required to build the underlying Cloud data-structure infrastructure. Most of these children have been using

Scratch for a while, and are familiar with other systems such as Minecraft, etc. but I felt it was worthwhile to share their perspectives on scaling. This is what one of the children responding to the survey had to say:

> "I was wondering how they managed to have everything hook up to one server and have it not crash repeatadely *[sic]*...or at least that's how I think cloud data works..."

Perspectives on privacy and safety

Cloud data-structures also led to a significant amount conversation around privacy and safety—especially in the five-day period when the Scratch 2.0 prototype was opened up to the public. A lot of the concerns expressed were around chat-rooms created with Cloud lists, and the safety aspects of having these kind of projects on the Scratch website[2]. However, there were concerns about privacy as well, especially in the context of the proposed username block, as the following comment shows:

> "Also, I have heard some people worry about the global cloud data variables. This feature has encouraged several users to consider creating high score leader boards for projects which should be a great way to increase competitiveness on game projects. The only issue is that the methods I have seen so far involve using the "ask" blocks to request a players name.
>
> Not only so some people (without really thinking) give their first name (which isn't too much of an issue) as an answer but some also give inappropriate words as jokes. The issue with both of these situations is that often the game doesn't have an option to remove a player from the leader board once added. So if I typed in my first name without thinking about it, after that everyone who views the project can see my

[2]The members of the team responsible for running the Scratch community website have traditionally refused to add any feature to the website for one-to-one or private communication capabilities.

name on the leader board. Further more to remove it I have to contact the owner of the project and request they remove it from the cloud data list. "

This again, reflects some of my previous discussions (chapter 5, section 5.4.1) on the user-name block, and though having an almost-anonymous user-id block somewhat defuses the privacy issue, some problems remain, such as the potential case of an user entering his real name in a Cloud data-structure by mistake.

7.2.4 Mental Models

Aside from trying to understand how children used the Cloud data-structure, and the learning outcomes, I was also interested how they saw beyond the "black-box" of the programming blocks (which hide all the complicated connection mechanism, data-storage, from the programmer using Scratch), and their mental models of the overall Cloud-data system. I present a couple of insights below:

Models for thinking about Cloud data-structures

Though I have framed the Cloud data-structure model as being the addition of an orthogonal property to data-structures in this thesis, an alternative model emerged during conversations with some of the participants of my studies, as well as through feedback messages in the forums. In this model, Cloud data-structures were seen as having some sort of "super-global" scope, applicable to each and every execution instance of a project. The primary difference between my framing and this particular model was the "lens" with which the children saw Cloud data-structures—the lens in my model was a new property of a data-structure, whereas the children used the more-familiar scope (global/local) of a data-structure as the lens.

This mental model may partially be a result of the fact that the initial prototypes had Cloud data-structures be global, and it would be interesting to see whether users stick to this model when both local and global data-structures in Scratch 2.0 can be "Cloud".

Models for thinking about internals of Cloud data-structures

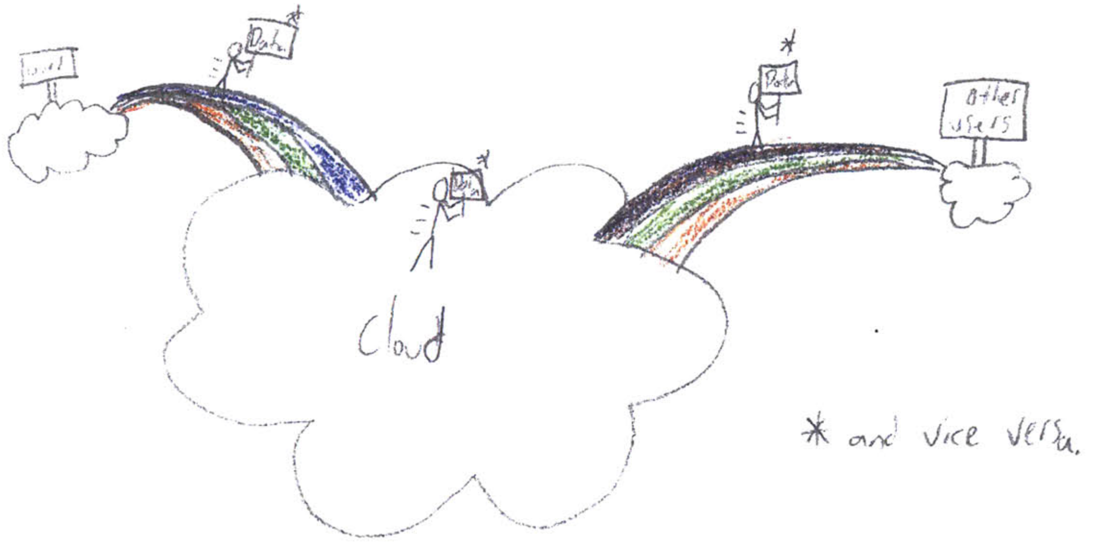


Figure 7-1: *Picture of how Cloud data-structures work, as drawn by a workshop participant*

At the conclusion of the first workshop, I asked the participants to draw a picture of how they thought Cloud variables and lists worked. Most drew models that were pretty close to reality—though in some cases, instead of drawing all the users as equal nodes, the participants gave one particular user priority status. In these diagrams, there was one particular node that was marked as "user" (or equivalent), while the rest of the nodes where marked as "other users" (or equivalent) figure 7-1. While the creator of a Cloud data-structure has a special status (described in chapter 5), the images did not specifically point out this special user as the creator, and in some ways, this might be an indication of the childrens' tendency to prefer a centralized model, which Mitchel Resnick covers extensively in his book *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds* [42]:

> "People seem to have a strong preference for centralization in almost everything they think and do. People tend to look for *the* cause, *the* reason, *the* driving force, *the* deciding factor . When people observe patterns and structures in the world (for example, the flocking patterns of birds or the foraging patterns of ants), they often assume centralized causes where none exist."

As I was writing this chapter, I realized that perhaps the best study of this system would only be possible when the larger Scratch community has access to the system. Given the system is about online data, and given that one of my core foci is generativity, the true range of possibilities would be apparent only after the "large, varied, and uncoordinated" member-body of the Scratch community starts to program with Cloud data-structures and use programs with Cloud data-structures. Though I highlight important and valid learning outcomes in this chapter, the contexts and settings I have used as testbeds for my system so far have been synthetic, and I am excited to imagine and think about what would happen when Scratch 2.0 is formally released, and when a much more diverse and larger audience starts to play with and use Cloud data-structures.

"And now for something completely different."

Monty Python

8 Future Directions & Conclusion

The area that I explore in this thesis forms only a small part of the space around learning with data. In this chapter, I suggest new areas to explore, and finally conclude with my reflections on the outcomes of my thesis.

8.1 Future Directions

I start off with possible incremental changes to the Cloud data-structure system, and then move on to other larger areas worthy of additional research.

8.1.1 Building on Cloud Data-structures

A potential extension of the design of Cloud data-structures is support for *inter-project sharing*. Scratch 2.0 introduces a new mechanism of sharing code snippets and multimedia assets between projects called the *backpack*. With the backpack, a Scratch 2.0 programmer can open up the code of a project, drag and drop code snippets and/or multimedia assets used in the project into a carousel like UI, and reuse them in another project. The backpack system can be utilized to establish links between projects, through Cloud data-structures. A Cloud variable or list from *project A*, transported through the backpack into *project B* can be used to share data between the two projects. Of course, this is only one possible way of achieving data-sharing, but sharing data between different projects seems like a natural next-step for enabling explorations of online data through Scratch 2.0, and a worthwhile topic to think about.

There has been calls from the advanced users of Scratch 2.0 to open up the Cloud data-structure protocol (and some have even managed to reverse-engineer parts of it), so that other programs can write to the data-structures, enabling Scratch 2.0 projects to "connect" to a diverse variety of software tools, and even, in some cases hardware extensions. While this certainly raises a lot of interesting possibilities, there is a need to be cautious here, so as to not to raise the low floor, and also to keep projects in the website understandable and devoid of dependancies on external, third party software.

8.1.2 Leveraging Mobile Devices

Perhaps one of the most exciting areas in the space of programming with data is in the domain of mobile devices, especially smartphones. Smartphones come with a wide range of sensors and they can also be interfaced with simple, low-cost sensor kits like pH probes, Geiger counters, etc. This enables one to easily collect data about a variety of topics through these phones. Moreover, with the presence of global positioning (GPS) capability in almost every smartphone, each data-point can be geotagged with precise location information, opening up an even wider variety of possibilities. As an example, after the Fukushima nuclear disaster in Japan [43], as a part of the citizen driven SafeCast[1] radiation data gathering network, a geiger counter smartphone accessory called iGeigie[2] was developed to enables anyone with an Apple iPhone to collect radiation data. The collected information was sent back to a centralized system that aggregated the submissions to create an interactive heat-map of radiation intensity all over Japan.

For explorations in the space of mobile devices and programming with data, a natural starting point would be App Inventor. While App Inventor already has components like TinyWebDB, and has support for data-oriented web services like Google Fusion Tables [44], an interesting approach would be to add Cloud data-structure support to App Inventor. To build upon this, it would be even possible to create bridges between App Inventor apps and Scratch projects, if the same data-store for Cloud data-structures is used, in a way similar to the inter-project data sharing system that I describe in the previous section. A potential technical challenge of building such a system is dealing with the wider variety of data-types App Inventor supports (Scratch 2.0 only supports numbers, strings and booleans), and trying to understand how that may work out with the Scratch bridge, but if such a system is made, it would be effectively enabling Scratch programmers to reach out to the wider physical world in a variety of new ways, starting from door-to-door surveys to long-term environmental monitoring projects.

8.1.3 Exploring Open Data

With a wide range of Open Data resources freely available online, enabling Scratch 2.0 projects to access Open Datasets would make possible another rich set of creative outcomes. Figure 8-1 shows mock-ups of a possible block system that enables access to structured Open Datasets (e.g. in the form of CSV files) online. As the figure shows, the block system enables the use of the same

[1]http://blog.safecast.org
[2]http://igeigie.com

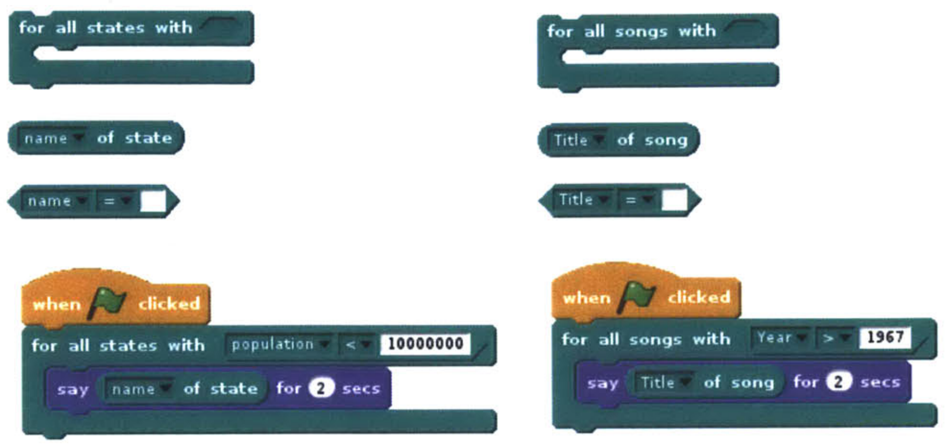

for all states with
name of state
name =
when clicked
for all states with population < 10000000
say name of state for 2 secs
for all songs with
Title of song
Title =
when clicked
for all songs with Year > 1967
say Title of song for 2 secs

Figure 8-1: *Mock-ups of Scratch blocks to access, filter and iterate over Open datasets*

block-grammar for accessing different datasets—in this case, US states, and songs released by the Beatles.

With such a system, it would not only be possible to access Open Data resources, but also, the design would allow for Scratch 2.0 community members to create and share their own datasets. The sharing can possibly happen through a *Block Exchange*, an online space where anyone would be able to upload datasets in a certain format and structure, allowing for others to (re)use the datasets in their own projects. A dataset on the *Block Exchange* system would be represented by a collection of blocks, which could be then "imported" into Scratch 2.0 projects, just as one imports images and sounds currently.

8.2 Final Remarks

To reiterate my earlier reflection, most of what I have covered in this thesis represent just the beginning of a longer journey. One of the contributions of this thesis is a simple design that enables novice programmers to program with online data. With the implementation of this design, within a very short period of time, a relatively small number of Scratch programmers created a diverse set of projects, covering a significant number of computational concepts and ideas around online data. As Scratch 2.0 is released to the public later in 2012, we can certainly expect to see the emergence of a larger variety of projects, and as with any generative technology and platform, as the designer, I have very little idea of the breadth of this variety. There is a challenge around the need for a large functional vocabulary, or a listing of the varied ways and contexts in which one can use Cloud data-structures, but given that a significant fraction of existing projects in the Scratch website are built upon other projects as "remixes" [45], I feel that a lot of this vocabulary would be gained from looking at existing projects in the community, as well as from sample projects that I put up in the website as "seeds".

Of course, there are potential problematic areas to deal with—moderation is certainly going to be something to think about, and there is some questions around privacy as well. Jonathan Zittrain warns us that "proponents of generative systems ignore the drawbacks attendant to generativity's success at their peril" [10], and for me, the only viable path to take at this juncture is to be aware of the problem areas, look at relevant projects and practices carefully, and from there, figure out interventions and workarounds. On the positive flip side of all this is, however, children actively engaging in conversations around issues such as privacy, and an increased amount of awareness on these topics. A glimpse of this can be seen in my findings in chapter 7, and this is certainly a rich research area that can be explored in greater depth in the future.

The ownership of online data is another open question. It is unclear at the moment what the reaction of the community is going to be when a project with crowdsourced data is remixed. Members of the community may react against the remixer getting access to the Cloud-data repository in the remixed project (which is the current behavior), something that may necessitate an attribution and crediting system for *data-remixing*, similar to the system [46] in place for project-remixing.

With so many interesting areas to explore, both for the novice programmer, as well as from the perspective of a researcher, I cannot but feel excited about what the future has in store.

A
Technical Implementation

A.1 Overview

The Scratch 2.0 authoring environment as well as the Scratch 2.0 interpreter is written in ActionScript 3.0, and both of them run via the Flash plugin in web-browsers. The server for Cloud data-structures is written in NodeJS, with MongoDB as the storage backend. When the interpreter loads a project with Cloud data, it establishes a TCP socket connection to the server and conducts a *handshake*, to register itself. Internally the server maintains a set of channels for each active project, and during the *handshake* process the server adds the connection to the relevant channel. During the *handshake*, the server also sends a copy of all the Cloud data-structure instances to the connecting client.

Table A.1: *Listing of the method calls used in the Cloud data-structure protocol.*

Method	Parameter[a]	Description
handshake[b]	projectId	Establishes connection to the server. The projectId parameter is used to add the connection to a server-side channel, used for later broadcasts.
set	variable name, value	Sets a variable to a the given value.
lset	list name, array	Sets a list to a the given JSON encoded array.
lappend	list name, value	Appends value to a list.
ldelete	list name, index	Deletes item at given index from a list.
linsert	list name, index, value	Inserts value at index of a list.
lreplace	list name, index, value	Replaces item at index with value in a list.

[a] Apart from the parameters listed in this table, each method call also includes the user-id, an authentication token, and the project-id.

[b] Not used for the fallback HTTP based protocol which uses a polling based system instead of broadcasts from the server.

Once the handshake process is complete, any operation on a Cloud data-structure instance in the interpreter results in a method-call message being sent to the server via the socket connection. The messages are JSON encoded, in a format inspired by the JSON-RPC specifications [47]. Table A.1 lists all the possible method calls that can be made to the server. An example message looks like the following:

{"method": "set", "name": "Votes for Scratch Cat", "value": "1"}

Once the server receives a message, it updates its internal data-store, and broadcasts back the method-call message to all the other clients in the channel, so that other instances of the interpreter running the same project get a push notification about the change in the data-structure. Figure A-1 illustrates the interaction between two instances of the Scratch 2.0 interpreter (running the same project) and the Cloud data server. In the diagram, after the handshake process, interpreter instance 1 invokes a *set* call, which is propagated to interpreter instance 2, followed by an invocation of *lappend* by interpreter instance 2, which is then propagated to interpreter instance 1.

A.2 Challenges

Concurrent operations and synchronization

The most significant technical challenges for this system fall in the category of concurrency and synchronization related issues. Dealing with these become especially challenging when Cloud data-structures are being updated in a tight loop (e.g. in a drawing project which stores x-y coordinates of the drawn lines in Cloud lists).

I try to deal with this only partially, by sending individual operations on data structures instead of passing the entire data structure around. Through this mechanism, even if operations are out of order, chances of complete data-loss in lists are lessened. This approach, does not however, work for Cloud variables.

Blocked non-HTTP traffic

A lot of school IT infrastructures have firewalls and proxies that block non HTTP traffic. In some cases, I have also observed built-in firewall applications in certain operating systems to block traffic to non-standard TCP ports. To deal with these scenarios, I implemented a fallback HTTP based protocol to emulate the socket-based protocol that I described before. For this fallback protocol changes to Cloud data-structure instances cannot be easily "pushed" to clients, and hence each client periodically poll a specific URL to get updated versions of the data-structures. The HTTP fallback server is also implemented in NodeJS.

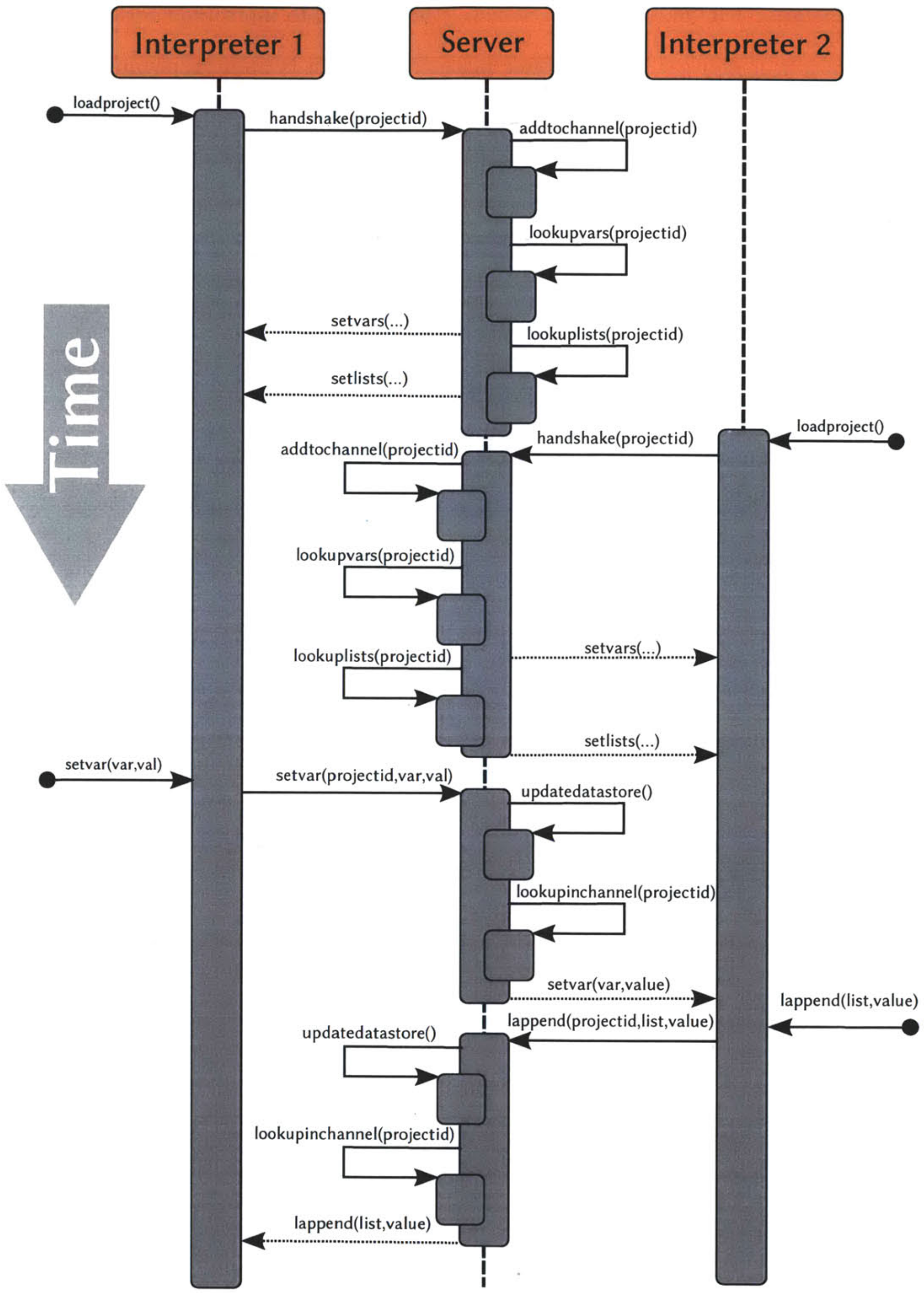


Figure A-1: *Instances of the Scratch 2.0 interpreter interacting with the Cloud data server*

B

Survey for Workshop Participants

I requested the participants of my first workshop to fill out an online survey at the conclusion of the second day of the workshop. Questions from the survey are listed in the next page. Apart from the survey, I also asked the participants to draw out, on a piece of paper, how they thought the Cloud data system worked. The results of the survey, as well as insights from the drawing exercise are incorporated in the findings that I have described in Chapter 7.

Cloud variables and lists – survey

We are interested to know more about your experiences with Cloud Variables and Lists. Your responses will help us as we work on Scratch 2.0. Thank you for your feedback and support!

- Your name:
  We will keep your name confidential.
- Your Scratch website username (if any): *[optional]*
  We will keep your username confidential as well.
- Grade you are in:
- For how long have you been using Scratch?
  ☐ Less than a year
  ☐ 1 – 2 years
  ☐ 2 – 3 years
  ☐ More than 3 years
- Apart from using Scratch, what are some of your other activities with a computer?
  Eg. playing games, programming in other languages (mention which one), etc.
- What surprised you when you used Cloud data?
- Did you find anything to be confusing?
- What are some of the projects where you would like to use Cloud data?
- Any other feedback? *[optional]*

Bibliography

[1] C. Anderson, "The end of theory: The data deluge makes the scientific method obsolete," *Wired Magazine*, vol. 16-7, July 2008.

[2] "Eric Schmidt: Every 2 days we create as much information as we did up to 2003." http://techcrunch.com/2010/08/04/schmidt-data/.

[3] P. Mell and T. Grance, *The NIST Definition of Cloud Computing*. National Institute of Standards and Technology, 53 ed., 2009.

[4] J. Naughton, "A manifesto for teaching computer science in the 21st century," *The Observer*, March 2012.

[5] A. diSessa, "Models of computation," in *User-Centered System Design: New Perspectives in Human-Computer Interaction*, pp. 201–218, Hillside, NJ: Lawrence Erlbaum Associates, 1986.

[6] A. Kay, "Computers, networks and education," *Scientific American*, vol. 265, pp. 138–148, September 1991.

[7] J. McCarthy, "Time sharing computer systems," in *Computers and the World of the Future* (M. Greenberger, ed.), pp. 220–248, Cambridge, MA, USA: The MIT Press, 1964.

[8] J. L. Zittrain, "The generative internet," *Harvard Law Review*, vol. 119, pp. 1974-2040, May 2006.

[9] J. L. Zittrain, "The personal computer is dead." http://www.technologyreview.com/news/426222/the-personal-computer-is-dead, November 2011.

[10] J. L. Zittrain, *The Future of the Internet and How to Stop It*. New Haven, CT, USA: Yale University Press, 2008.

[11] S. Papert, "Hard fun," *Bangor Daily News*, 2002.

[12] M. Resnick and B. Silverman, "Some reflections on designing construction kits for kids," in *Proceedings of the 2005 conference on Interaction design and children*, IDC '05, (New York, NY, USA), pp. 117–122, ACM, 2005.

[13] J. Howe, "The rise of crowdsourcing," *Wired Magazine*, vol. 14-6, June 2006.

[14] S. Papert and C. Solomon, "Twenty things to do with a computer," AI Memo 248, MIT AI Lab, 1971.

[15] S. Papert, *The Children's Machine: Rethinking School in the Age of the Computer.* New York, NY: Basic Books, 1993.

[16] S. Papert and I. Harel, "Situating constructionism," in *Constructionism*, pp. 1–11, Ablex Publishing Corporation, Norwood, NJ, 1991.

[17] S. Papert, *Mindstorms: Children, Computers, and Powerful Ideas.* New York, NY: Basic Books, 1980.

[18] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: programming for all," *Communications of the ACM*, vol. 52, pp. 60–67, November 2009.

[19] J. Dewey, *Democracy and Education: An Introduction to the Philosophy of Education.* The Macmillan Company, 1916.

[20] R. Tagore, *Creative Unity.* The Macmillan Company, 1922.

[21] P. Freire, *Pedagogy of the Oppressed.* Penguin, 1970.

[22] J. Lave and E. Wenger, *Situated Learning: Legitimate Peripheral Participation.* Cambridge University Press, 1991.

[23] S. Papert, "What's the big idea? toward a pedagogy of idea power," *IBM Systems Journal*, vol. 39, no. 3.4, pp. 720–729, 2000.

[24] H. Ishii, A. Mazalek, and J. Lee, "Bottles as a minimal interface to access digital information," in *CHI '01 extended abstracts on Human factors in computing systems*, CHI EA '01, (New York, NY, USA), pp. 187–188, ACM, 2001.

[25] 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*, vol. 2, pp. 48–54, February 2011.

[26] R. Waters, "A method for analyzing loop programs," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 237–247, May 1979.

[27] E. Tufte, *The Visual Display of Quantitative Information.* Cheshire, CT: Graphics Press, 1983.

[28] B. J. Fry, *Computational Information Design.* PhD thesis, Massachusetts Institute of Technology, 2004.

[29] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs.* Cambridge, MA, USA: MIT Press, 1985.

[30] A. Monroy-Hernández and M. Resnick, "Empowering kids to create and share programmable media," *Interactions*, vol. 15, pp. 50–53, March 2008.

[31] M. Atkinson, P. Bailey, K. Chisholm, P. Cockshott, and R. Morrison, "An approach to persistent programming," *The Computer Journal*, vol. 26, no. 4, pp. 360–365, 1983.

[32] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan, "Making data structures persistent," *Journal of Computer and System Sciences*, vol. 38, no. 1, pp. 86–124, 1989.

[33] D. Wolber, H. Abelson, E. Spertus, and L. Looney, *App Inventor: Create Your Own Android Apps.* Sebastopol, CA: O'Reilly Media, Inc., 2011.

[34] T. Stern, "NetScratch: A networked programming environment for children," Master's thesis, Massachusetts Institute of Technology, 2007.

[35] "Amazon Simple Storage Service (Amazon S3)." http://aws.amazon.com/s3/.

[36] J. Maeda, *The Laws of Simplicity.* Cambridge, MA, USA: The MIT Press, 2006.

[37] M. Resnick, "MultiLogo: A study of children and concurrent programming," Master's thesis, Massachusetts Institute of Technology, 1988.

[38] A. J. Perlis, "Special feature: Epigrams on programming," *SIGPLAN Notices*, vol. 17, pp. 7–13, September 1982.

[39] K. Brennan, A. Valverde, J. Prempeh, R. Roque, and M. Chung, "More than code: The significance of social interactions in young people's development as interactive media creators," in *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2011* (T. Bastiaens and M. Ebner, eds.), (Lisbon, Portugal), pp. 2147–2156, AACE, June 2011.

[40] S. A. Robila and J. W. Ragucci, "Don't be a phish: steps in user education," in *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, ITICSE '06, (New York, NY, USA), pp. 237–241, ACM, 2006.

[41] M. Minsky, "Introduction to LogoWorks," in *LogoWorks: Challenging Programs in Logo* (C. Solomon, M. Minsky, and B. Harvey, eds.), New York, NY, USA: McGraw-Hill, Inc., 1986.

[42] M. Resnick, *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds.* Cambridge, MA, USA: MIT Press, 1994.

[43] E. Strickland, "Explainer: What went wrong in Japan's nuclear reactors." http://spectrum.ieee.org/tech-talk/energy/nuclear/explainer-what-went-wrong-in-japans-nuclear-reactors, March 2011.

[44] H. Gonzalez, A. Y. Halevy, C. S. Jensen, A. Langen, J. Madhavan, R. Shapley, W. Shen, and J. Goldberg-Kidon, "Google fusion tables: web-centered data management and collaboration," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, (New York, NY, USA), pp. 1061–1066, ACM, 2010.

[45] A. Monroy-Hernández and B. M. Hill, "Cooperation and attribution in an online community of young creators." Poster at the ACM Computer Supported Cooperative Work Conference (CSCW '10), 2010.

[46] A. Monroy-Hernández, B. M. Hill, J. Gonzalez-Rivero, and d. boyd, "Computers can't give credit: how automatic attribution falls short in an online remixing community," in *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, (New York, NY, USA), pp. 3421–3430, ACM, 2011.

[47] "JSON-RPC 2.0 specification." http://www.jsonrpc.org/specification, December 2011.