Math is difficult. Year after year, many students are turned away from the subject as a result of its intimidating nature. There have been attempts at making the subject friendlier, especially to younger students, but each approach compromises on some aspect of mathematics for the sake of seeming friendlier to students. The application Algebra Touch is a good example of this compromise. It successfully makes the act of solving simple algebra equations less tedious but does so by limiting the kinds of equations the user can work with. Here, I present Shoreline, the beginning of an attempt at math software that can be expanded to let users solve a wide variety of math expressions while making the subject less intimidating to its users. This paper describes the work's motivation, influences, initial implementation, and evaluation.

1. Problem

There are students who don't want to do math because it is difficult, and students who don't want to do math because it isn't fun. Most students avoid math for both reasons. Shoreline attempts to remedy this problem by identifying the source of math's deterrents. I propose that one reason for mathematics' difficulty is its cryptic nature. When a student first looks at a math equation and is asked to "solve for x", they are only able to carry out the assigned task if they know what it means to solve for a letter in an equation and the rules that govern the manipulation of an equation. As the student moves up through different math curricula, the tasks expected of them to accomplish become more complicated, and the rules that allow the student to make the correct manipulations become more numerous. If they fail to commit a rule to memory, they are unable to even attempt to solve the problem. As they stare at a math problem, staring back at them is a collection of symbols whose meaning is only vaguely understood. This is inherent to all kinds of mathematics. In order for the student to reach the point where they can remember what is possible with their cryptic set of symbols, they must work with the symbols until they are no longer cryptic, which often takes a lot of time and causes a lot of tedium.

A possible reason behind math's tedious nature is paper and pencil's status as the accepted medium of work in mathematics. When manipulating a simple math equation, the student starts at the top of the page, determines what rule to apply that will change the equation into a new form, and repeats the process until the equation is in the form that they want. A more explicit algorithm for solving an algebra problem is as follows:

1) Determine desired form of expression
2) Choose an algebraic rule that may or may not lead get the student closer to the desired form
3) If it is clear that the rule will not lead the student towards the desired form, go to (2)
4) Use chosen rule to manipulate expression into new form
5) If it is clear that the past rules used will not lead to the desired form, start back to the form of the expression that might lead to the desired form and go to (2)
6) If the current expression is not in the desired form, go to (2)
7) The equation has been solved

All of these steps are required in order for the student to properly think through the problem. Each kind of medium for solving math equations handles step (4) differently. Using paper and pencil, students have to write down the equation's new form on every iteration of the algorithm. This isn't an issue when working with simple equations, but the tedium the student feels increases proportionally with the size and complexity of the equations involved.

In order to invent a system that makes solving complex equations less tedious, one must first create a system that makes solving simple equations less tedious. Initially, Shoreline will be developed with the goal of allowing users to solve problems they might encounter in a beginner algebra course, where there are few terms to work with and the only operators involved or addition, subtraction, multiplication, and division. The target audience for Shoreline is made up of Algebra I students who are capable of learning the material, but don't want to work on it because of the tedium involved. An example of a student that might benefit from Shoreline is presented below.

Freddy is a high school freshman with a failing grade in math. He spends most of his days on his computer playing video games or watching other people play video games. He doesn't need to do most of his homework at home since he often finishes it at school. His favorite games are puzzle games. Math used to be his best subject and he would do most of the problems in his head. He thought it was fun. Then Freddy took his first Algebra class. At first, he remained successful in keeping the problems in his head, but they eventually became too big to keep track of everything at once. Freddy made an attempt to solve the problems on paper like his teachers expected him too, but it took too longer and didn't feel worth his time. Math had lost its fun for Freddy. He didn't care that his grades started dropping but he knew his mom would, so he didn't tell her about it.

Freddy's mom, Olivia, raises Freddy on her own and works a full-time job. She is thankful that Freddy has done so well in school despite not being around to tell him to do his homework. In order to get a higher paying job, Olivia started taking classes at the local community college at night and on weekends. This has stretched her budget pretty thin, and she and Freddy are practically living paycheck to paycheck. She is about to enter her final quarter and has already gotten a job that starts after she graduates. She is comforted by the knowledge that she won't have to pinch pennies for much longer. Olivia checks in on Freddy when she gets home from work every day and asks if he has finished his homework. Freddy's answer is always yes.

One day, Freddy's transcript arrived in the mail. Olivia thought this was odd since they had never received his grades in the mail before, but she didn't think much about it and opened up the envelope. She was horrified to see that he was failing his Algebra class. After a quick email to Freddy's Algebra teacher where she asked why his grade was so low, she went to Freddy room to find him sitting at his computer. He was playing Sokoban. Olivia, unsuccessful in hiding her frustration, asked Freddy why there was an "F" listed where his Algebra grade should be. Freddy told her that there must be a mistake since he'd been completing all his homework and doing well on his tests. At that moment, Olivia's phone made a noise. She took it out of her pocket and turned on the screen, where she saw Freddy's Algebra teacher's reply. It turns out Freddy stopped turning in his homework a month earlier, and his test scores only dropped from then on.

Olivia set up a meeting with herself, Freddy, and Freddy's teacher. The goal of the meeting was to establish a plan of action that would ensure Freddy passed Algebra. Freddy's teacher told them that Freddy wouldn't have to make up the homework he didn't turn in if he *retook and was able to earn a B+ or higher on every test he had taken from when he stopped turning in homework. Olivia asked the teacher how Freddy could motivate himself to do his work. Freddy's teacher then asked Freddy what he thought would help. Freddy said he'd be able to motivate himself and didn't need help doing so. At the end of the meeting they scheduled the makeup tests. The first one was in a week, and they were all confident Freddy would do well on it since it tested content that was easy relative to the rest of the course.*

Since the meeting, Freddy had been studying in the living room everyday by the time his mom came home. Three days before the first make-up test, Olivia asked Freddy if he's prepared. Freddy responded to her question, saying that he thought he could pass the first test, but he didn't feel like studying for the others. Instead he would just retake Algebra next semester. In shock, Olivia went to her laptop and hired an online tutor that Freddy was to meet with for a couple hours every week. It's expensive, but she figured it's worth it if it prevents Freddy from having a failed class on his transcript. An hour later, Olivia received an email that registration for her last quarter at the community college was coming up. There's no way she can pay for next quarter's classes and hire a tutor for Freddy. She decides to put her son's interests before her own and keeps the tutor.

Shoreline would help students like Freddy by removing some of the barriers keeping students from wanting to work on math problems, though many of these barriers will remain. Shoreline will try to prevent the tedium that early algebra students experience and reduce the frustration that is associated with not being shown what can be done with an algebra equation. I believe that both barriers can be lessened through the use of alternatives to pencil and paper work in algebra. Freddy is based on people I've met prior to this directed study as well as interviews I conducted while evaluating the app Algebra Touch.

2. Algebra Touch

Algebra Touch is an iOS app created by Regular Berry software. According to its description on Apple's App Store, it's meant to "…refresh your [algebra] skills using touch-based techniques built from the ground up". I was able to find one academic paper involving Algebra Touch, but it doesn't explore the effectiveness of the app in refreshing someone's algebra skills, or as a replacement for solving algebra equations on paper. Nonetheless, Algebra Touch is the closest application I could find with any notoriety that partially accomplishes what I set out to accomplish with Shoreline. Because of this, I found it helpful to heuristically evaluate Algebra Touch using Nielson's Heuristics, as well as conduct a formative evaluation of the software with the help of users who feel a similar way towards mathematics as Freddy does.

2.1. Heuristic Evaluation

Algebra Touch receives a passing grade for most of Neilson's Heuristics. The app makes it clear when the user makes a mistake. For example, when the user taps an operator in Algebra Touch, the terms to its immediate left and right are evaluated according to that operator. If the surrounding terms aren't like terms with each other, the expression shakes, indicating to the user that they made an error. A similar affect occurs when the user taps anything else on screen when they shouldn't. This falls in line with the "Visibility of System Status" heuristic. I couldn't find any examples where this heuristic was violated.

Neilson's second heuristic, on having the system match the real world, can be applied to an algebraic manipulation app such as Algebra Touch if the map between the app and the world occurs between how algebraic expressions are manipulated in real life, and how they are manipulated within the app. Evaluating Algebra Touch based on this mapping would cause the app to fail in meeting this heuristic. Apps like Algebra Touch or Shoreline are being created to offer alternatives to the "real world". To many students, working with math expressions in the real world feels like pulling teeth, and recreating that sense in an application meant to make people feel less intimidated by algebra would run counter to the app's goals. Algebra Touch passes this heuristic in the sense that users familiar with solving simple algebra problems are familiar with the symbols it presents and some of the gestures it provides to the user. There are downsides to thinking about the action that turns the equation $a = b + c$ into $a - b = c$ as "moving" the b to the other side and negating it, but it's the way a lot of former algebra students think about the action. Algebra Touch is missing ways to easily switch out different ways of thinking about these actions, so it's mapping between itself and the "real world" of mathematics is set in stone.

The "User control and freedom" heuristic requires an app give the user an "escape" if they mistake is made. The most important kind of mistake a user in Algebra Touch can make is accidentally applying the wrong algebra rule and needing to go back. For this, there is an "undo" button located in the top right corner of the screen, but it is mostly unnecessary, since every manipulation a user makes to an expression is reversible. Reversing the step by hand often requires as much or less thought than tapping the undo button requires, but some users may find the undo button helpful. Some steps towards the end of Algebra Touch's sequence of problems cannot be reversed by hand without a lot of effort, requiring the undo button.

Algebra Touch scores moderately well in the "flexibility and efficiency of use" heuristic. There are a few hidden ways of manipulating expressions that offer alternatives to manipulations taught by the app. For example, by tapping the equals sign in an equation, the user can add, subtract, multiply, or divide the same value from both sides of the equation, greatly opening up the possible equations one can work with. The existence of this feature is consistent with other ways of manipulating equations, as the user is able to tap on any other term or operator in the expression and have something happen to it. Algebra Touch also offers a way to input custom expressions so that the user can try out their own problems.

The final set of heuristics in which Algebra Touch does well in concern minimal design, helping users diagnose and recover from errors, as well as providing documentation to users who seek it out. However, the app falls on its face in the fourth and sixth Heuristic.

Algebra Touch follows the "Consistency" heuristic well when looking at interface elements that are common in other applications. When I go to press a button in the app, or type in a new expression, everything behaves like it would in more familiar software. The largest shortcoming of Algebra Touch, however, is in its lack of internal consistency. For example, most problems in the app will not allow you to drag a term on top of another term, but if the user drags a term that is being multiplied by another on to a different term that is being multiplied by another, the distributive property is applied. To better illustrate, in the expression $a + b$, dragging the b onto the a will not accomplish anything. The b will simply slide back to its original position. However, if the user were working with the expression $a * b + c * b$ and dragged the b being multiplied by the a to the b being multiplied by the c, the distributive property will be applied, and the equation will become $(a + c) * b$. This action is neither discoverable nor taught to the user. It does, however, follow the rule that every transformation of an equation be reversible.

Another example of a break in Algebra Touch's internal consistency is in it's handling of division of exponents. The user is originally taught to cancel out common factors in the numerator and denominator of a rational expression by factoring out the common factors in each term, then drawing a line through the common factors in the numerator and denominator to cancel them out. For example, the user would tap the 6 in the expression $\frac{6}{2}$, turning it into $\frac{3*2}{2}$, and finally draw a line through the 2 in the numerator and the 2 in the denominator to end up with 3. A second, hidden way of achieving the same goal is by simply drawing a line between the original numerator and denominator, which then automatically factors both numerator and denominator, allowing users to cross out the common factors like they do using the other method. This replaces the step where the user has to tap on the 6 to turn it into $3 * 2$. This action can be seen as an accelerator. Having been taught this, when the user is presented with the expression $\frac{x^8}{x^4}$, they expect to be able to draw a line through the numerator and denominator and end up with $\frac{x^4 * x^4}{x^4}$. Algebra Touch doesn't allow this.

Nielson's sixth heuristic stresses "recognition rather than recall". This means that a user must be able to know what to do based on the options that are laid out before them, and not on what they remember being able to do in the past. Paper- and pencil-based mathematics would fail this heuristic, since the user is required to remember what they are allowed to do with an equation. Algebra Touch fails for the same reason. At the beginning of a section, the user is shown a short recording of a circle floating around that taps and drags parts of the expression. This is supposed to show the user how to apply the new concept the app is trying to teach for that section. While working on a problem, the user is allowed to tap the "Help" button in the top right corner of the screen and watch the same recording play. All this recording does is show the user what the app wants them to do, and not everything that the user can do. As a result, the user misses out on alternative solutions to the problem, which is important when learning how to solve algebra problems. It can be argued that hiding all the ways the user can manipulate math expressions is important because it mimics doing math in the real world, but the purpose of mathematics is not memorization. Solving problems is.

2.2. User Evaluation

The ideal candidate for this evaluation would have been a high school Algebra I student, since they are the population I am targeting with Shoreline, but I had to find another population of users to evaluate due to ethical restrictions. The evaluation subjects I decided to look for were Northeastern students who didn't like mathematics. If an app were able to help members of this population tolerate doing math problems, the same app might keep beginner algebra students engaged long enough to learn something.

The three "math-hating" candidates I found all live in the same apartment and had friends entering and exiting the room while the evaluation was taking place. When a new person entered, they always asked what we were doing, and later expressed interest in being a part of the evaluation. All of the last-minute subjects actively enjoyed math, which went against the criteria I set for the evaluation subjects, but I figured they might give more insight into what Algebra Touch does right and wrong.

Of the three "math-haters" I found, one was a second-year political science major, another was a first-year economics major, and the last was a fourth-year computer science major. Before the evaluation, I asked each of them seven questions about their relationship with mathematics up to until now. The questions asked are as follows:

1) Why don't you like math?
2) When did you start disliking math?
3) If you ever liked math in the past, why did you like it?
4) When you are presented with a problem and are unable to solve it, what do you do?
5) What do you like about your major?
6) Why do you think some people are better at math than others?
7) Before high school, what did you want your career to be?

From the first three questions, I learned that the subjects enjoyed doing mathematics when it was easy for them and started disliking it when it's increasing difficulty lowered their confidence levels. In question (4), all of the subjects said they were likely to reach out for help when they weren't making progress on a math problem, and each expressed a willingness to give up on the problem if it wasn't a problem they were required to solve. Two of the three math-haters were in majors that required some sort of mathematical problem solving. I was surprised to hear the computer science major express satisfaction in solving difficult problems in her subject and speak highly of the logic and creativity associated with computer science. When asked how this was different than mathematics, she responded by saying there was no creativity in mathematics; that it feels like learning to copy things down. The economics major said that she chose her major for the job prospects, and that she liked microeconomics more than macroeconomics. As a non-econ major, I've always been under the impression that introductory microeconomics classes require more math than intro macroeconomics classes, so this answer struck me as odd. I expected their answers to question (6) to mention the inherent math ability students who are good at math, but instead each of the subject's answers mentioned that confidence and effort were required to be good at mathematics. I was only able to ask question 7 to two of the three math haters. They both said they wanted to be some form of artist. The economics major wanted to be an actress and the computer science major wanted to be a writer. They both had a change of heart when they learned of their job prospects. I used these questions and answers to form Freddy's persona, which helped me more clearly define the problem I was trying to solve with Shoreline.

Algebra Touch is split up into eight sections, titled "Simplify", "Like Terms", "Multiply and Add", "Factors", "Elimination", "Equations", "Distribution", and "Exponents". For each subject, I cleared the progress that had been made working through these sections and handed each user the iPad running Algebra Touch. One of the three math haters attempted to manipulate the "$2 + 2$" expression that the user is supposed to tap before starting the Simplify section, which you cannot do. Simplify has the user tap the operations between terms to simplify the expressions. For example, the expression $2 + 3 + 4$ becomes $2 + 7$ after the user taps the $+$ between the 3 and the 4. The subjects said that their minds were blank when performing this action, as all they had to do to was tap every operator they saw.

The Like Terms section had the subjects drag terms around the screen until they were next to other like terms, then tap the operator between the terms to combine them, like they did in the simplify section. Because of the similarities between this section and the previous section, the subjects didn't comment on it much.

Multiply and Add reinforces the operator precedence between multiplication and addition, having the user go through problems where they may be tempted to add then multiply instead. This was the first section in which subjects accidentally manipulated expressions in ways they didn't intend to. For example, factoring is a topic covered by a later section, but the user can factor any term in any problem by tapping on it, no matter the section they are in. Only one math-hater ran into this problem with factoring and they were able to undo their action easily, but they were noticeably confused when they made their error.

The first notable consistency issue within Algebra Touch occurs in the Factoring section. For each problem, the user is presented with a number, like 12. Once they tap the number, a list of ways to factor the number pops up above it, for example "$2 * 6$, $3 * 4$". The user is then expected to tap one of these factored forms and repeat the process until the original number is turned into its prime factorization, which would be $2 * 2 * 3$ in this case. The lack of consistency is in what the app sets as a goal for the user. In the first three sections, the user is trying to reduce the given expression until there are as few terms on screen as possible. Factoring expects users to factor the number until it is completely factored so that a maximum number of terms are on the screen. All of the math-haters got into the habit of factoring the expression until they thought they had finished, then clicking the "Next Problem" button, which only generates a new problem instead of advancing the user towards the next section. Two of the three figured out the goal with enough trial and error.

Factoring is a prerequisite to the Elimination section, where users are expected to turn an expression such as $\frac{6}{2}$ into 3 by factoring out a 2 from the numerator, then drawing a line through the $2s$ in the numerator and denominator to cancel them out. One of the users started off expecting to move on to the next problem after factoring every number they saw on screen. An issue among all of the subjects, including myself, occurred when they attempted to draw the line to cross out the numerator and denominator. When starting the line, each subject would accidentally grab the numerator and drag it down, often repeatedly before slowing down and successfully drawing the line. There is also a hidden feature in this section which every subject managed to discover. This feature is detailed in the heuristic evaluation.

Each subject described a "flow" that they experienced once they reached the Equations section. At this point, they had used the app long enough to be comfortable with the basic manipulations. The only new manipulations in this section are extensions to the dragging manipulations that allow users to commute terms that are being added or multiplied. When a term that is being added to another term is dragged across the equals sign, the sign of the term will flip before it is dropped on the other side. Similarly, when a term being multiplied by another term is dragged across the equals sign, the term will divide the other side of the equals. "Flow" refers to the small amount of brain activity required to solve a problem. This is a problem for any math-based application.

Up until the Distribution section, each section relied on two gestures to manipulate an expression: dragging a term and tapping an operator or a term. The Distribution section adds a third gesture. In order to distribute a term across two terms being added together, like in the expression $a * (b + c)$, the user is expected to tap the multiplication operator, where a slider pops up at the bottom of the screen. If the a were to the right of the $(b + c)$, the slider will go from left to right, and it would go from right to left if the a were on the left, like in the example given. Every subject tried to distribute the a by dragging the term onto the addition operator separating the a and the b. They did this even after seeing the recording showing them how to properly solve the problem. The slider is so inconsistent with the rest of the app that the subjects didn't notice that it was there.

Only one user finished the Exponents section. This is partially due to fatigue that set in twenty to thirty minutes into working through the sections, but I blame Algebra Touch's handling of exponents more than anything else. There are two kinds of problems in this section: problems of the type $x^n * x^m$, where the user is expected to manipulate the expression into x^{m+n}, and problems of the type $\frac{x^n}{x^m}$, where the user is expected to manipulate the expression into x^{n-m}. In both cases, the user first tap one of the x^ns, like they would do to factor a number, but instead of being greeted with a list of possible factors for x^n, two green bars appears next to x^n, then animate by "pulling" each x out of x^n until the user is left with $x * x * \ldots * x$ for a total of n xs being multiplied by each other. In the case where users are trying to work the equation to x^{n+m}, they're supposed to expand the second term in the expression, then recombine all the xs into one term by dragging one of the green bars to the opposite side of the screen. Or at least I think that's how it works. None of the subjects thought this was clear. Up until this point, the users all skipped the short recording at the beginning of the section. Now they were all pressing the help button to watch the recording play, tapping on the screen to make it go away when they felt they knew what was going on, then pressing the help button again when they realized they didn't know what was going on. One of the subjects was able to get through this section through extensive trial and error. The other two gave up within the first few problems.

2.3. Algebra Touch Conclusion

Major themes pulled from the Algebra Touch user and heuristic evaluations include work done by the user vs. work done by the app, consistency, and extensibility.

The goal Algebra Touch states for itself in its App Store description is to help refresh the user's algebra skills. The goal justifies the lack of thought necessary in solving problems within the app but had me and the math-haters asking what the point of the app was. If someone wants to be comfortable doing algebra – which any app that claims to refresh algebra skills should help people accomplish – they should be able to think through the problems themselves. This means avoiding the "flow" felt by the math-haters by designing an app that doesn't think for its user. The difficulty comes in designing an app that removes this flow while ensuring that those who aren't fond of math still want to use the app. Algebra Touch does a good job at achieving the latter goal, as all my subjects stayed friends with me after making them use it, but it does a poor job with the former goal.

Algebra Touch falls on its face in terms of consistency. Consistency is important for an app that seeks to do what Algebra Touch is trying to accomplish because it is more important for users to remember how to do math than it is for users to remember how to use the app. Ideally, an app that teaches, refreshes, or just shows someone how to do algebra would have gestures for manipulating expressions that are internally consistent so that when a new topic is introduced to the user, they can thoughtlessly figure out what gestures are needed to manipulate that topic and focus on understanding the topic in-depth. Algebra Touch relies on dragging and tapping terms and operators for its gestures until the Distribution section, where it introduces a gesture that isn't consistent with anything that came before it. The Exponents section takes this lack of consistency one step further by taking term tapping, an established way of manipulating an expression, and changing what it does.

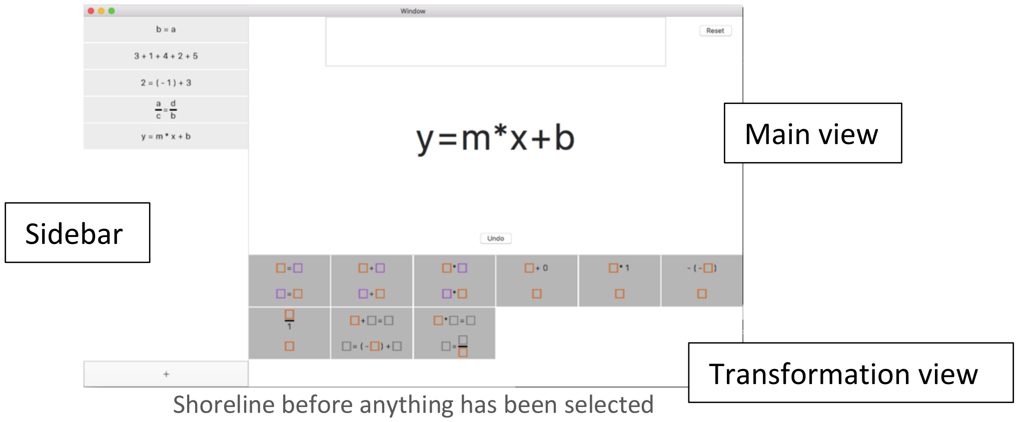
I suspect that Regular Berry originally designed Algebra Touch to stop after the Equations section. When they went back to add Distribution and Exponents, they had trouble coming up with ways to distribute terms and work with exponents that fit within the app's established conventions, so they added new conventions. Because mathematics is a large, ever-expanding subject, any app that attempts to allow its users to work with mathematics like they would on paper must be expandable while staying internally consistent. An expandable math application wouldn't run into the same problems Algebra Touch does with distribution. Complex new concepts should be expressible using the same elements used to make up the most basic concepts within the app.

3. Design

Shoreline's main goal is to allow its users to solve simple algebra equations on a computer without the tedious nature of solving the same problems with pencil and paper. Shoreline must also accurately recreate the thought process of students who are working through the same kinds of problems on paper. In the future I plan to expand upon what Shoreline can do so that it can handle more complicated algebra problems, which will require it to have a simple system to add and remove features without breaking consistency with older features.

I claim that the tedium that occurs when doing algebra on paper is due to the need to rewrite the expression for every manipulation the student makes. Therefore, Shoreline must allow its users to change the state of an expression without entering the new form of the expression. In order for the user to see their past work, Shoreline must give the user easy access to an expression history that displays what rules were applied to transform the expression.

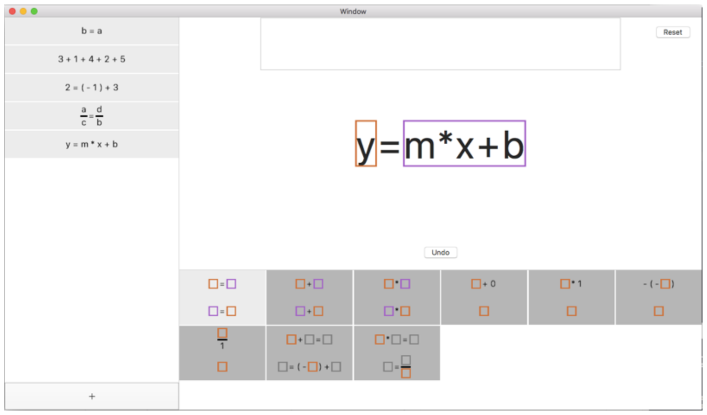
Recreating the thought process behind solving a problem on paper is difficult to do and requires an amount of time and money not available to me for this directed study, so the reasoning for the solution I give here is likely lacking. In the task algorithm I give above, the user needs to identify the form of the expression that they want to end up with, and they need to identify what rules they can apply to the expression that fit that form and its current form. The process is similar to pattern matching – the user identifies what rule they need, and searches for patterns in the expression that match one side of that rule. My attempt at recreating this process is described in more detail below.



Shoreline before anything has been selected

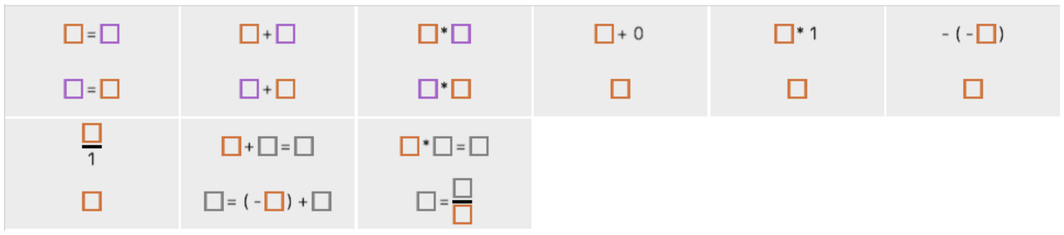
Shoreline is made up of a sidebar, transformation view, and main view. The sidebar gives the user access to other expressions that they can work with. The "+" button at the bottom of the sidebar can be clicked, which allows the user to enter another expression they might want to work with. The transformation view shows every rule that can be applied to the current expression to transform it into another expression. The colored boxes tell the user how they should select the expression in the main view in order to apply the rule. The main view is where the expression the user is manipulating lives. The banner-ad looking box at the top of the main view is where the expression history can be viewed. To the right of the history banner is a reset button, which allows the user to reset the expression to its original state. Below the displayed expression is an undo button so the user can return to the last state of the expression.

The heart of interaction in Shoreline occurs between the main view and the transformation view. If the user wants to manipulate the equation according to a rule, they first select the relevant parts of the expression, causing the rules that are possible to apply to light up, allowing the user to click on the rule to apply it. In the image below, the left and right side of the current expression are selected. The only rule that can be applied to an expression that is selected this way is the rule that swaps each side of an equal sign. Selections are made by dragging the mouse over parts of the expression. Multiple selections are made by command-dragging across a different part of the expression.

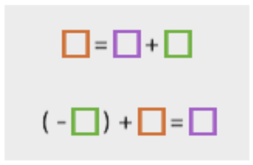


Selecting parts of the expression changes what rules the user can apply

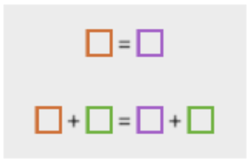
Each rule corresponds to an algebraic identity. Each rule is reversible, so the user can select the current expression in a way that matches the top or bottom of the rule and it will still match. Whichever side of the rule isn't selected is what the current expression is transformed into once the rule is applied. The colored squares represent any expression, as long as the expression represented by a square of a specific color is the same for all squares of that color in the rule. For example, applying the rule to the expression and selection in the above image will result in the expression $m * x + b = y$, since the orange and purple boxes swapped sides. The gray box is the wildcard box, matching any expression.


Initial set of transformations

Of the five rules I started with in the paper prototype, I changed two of them to better reflect how the user is expected to interact with the application and added three more as I was testing the app on myself. The three that I added were the last three rules in the top row of the image above. The two rules that replaced others in the paper prototype are the last two rules in the bottom row of the image above. The second to last rule replaced the rule below. My paper prototype testers were able to figure out what this rule did but were unable to determine how to use it. I had planned to take order of selection into account, so that orange would be selected first, purple second, and green third, which would tell the application exactly how to apply the transformation. There was nothing in the paper prototype indicating that selection order mattered, and the comment I kept getting for this type of selection was "that's dumb", so I scrapped it.


Bad rule

I originally planned on giving the user the ability to input their own expressions, which would make rules that added expressions to the current one possible, like the one below. This rule corresponds to the identity $a = b \Longleftrightarrow a + c = b + c$, with a corresponding to orange, b to purple, and c to green. If users select the current expression according to the bottom of the rule, the rule would get rid of the expression selected in green. If the user selected the current expression according to the top of the rule, the green box would show up in the new expression as textboxes with a green border. The user would then be able to input whatever valid expression they wanted into one of the text boxes and it would be reflected in the opposite one. Once they hit enter the user can then continue manipulating the expression as they were before. This kind of rule didn't make it into Shoreline because of time constraints, but it's something I'd still like to add.



A rule that allows for the creation of expressions

When a part of the expression is selected, the list of rules changes in order to give the user a better sense of how the selections map to the available rules. This can be done in a number of different ways. The first way that I considered had the rules reflowing in the view so that rules matching the selection "more" than others would be closer to the top of the list of rules. I found that a binary match/doesn't match filtering system was easier for me to reason about and my paper prototype testers didn't notice a difference. The second way of changing the list of rules would remove rules from the list that didn't match and keep the ones that did in the view. There's a heuristic somewhere that says UI elements shouldn't disappear. It also doesn't achieve anything more than would be achieved if unmatched rules were locked, so I didn't implement this either. The third way of changing the list of rules that I considered would have all matching rules flow to the top of the list, and all the others flowing to the bottom. I think this is the best approach because it shows the user what rules they are allowed to use while keeping the usable rules closer to the top of the list. The approach I went with only disables some rules from being used without reflowing the transformation view's elements, but I think having them reflow is necessary if the number of transformations exceeds twelve, the maximum number of transformations the app can have before the user has to scroll down to see all of them. There is still the issue of having more than twelve matching transformations for a given expression selection, which would require some kind of closeness metric for determining how much a selection matches a rule. Before I think about that problem in depth there need to be more than twelve possible rules to match against.

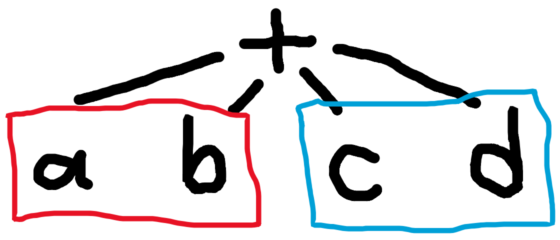
4. Implementation

Shoreline was originally meant to run on an iPad, but it didn't take much convincing for me to target a mouse and keyboard instead. Touch makes selection difficult, as evidenced by iOS's solution to the problem which has the user hold down on the text they want to edit, they drag a tiny handle to increase/decrease the selection while a virtual magnifying glass helps you see what your finger is covering up. Writing Shoreline for the desktop didn't get rid of the difficulty in implementing user-friendly selection, though. I hadn't nailed down the ideal method before it came time to start implementing, so I went with the quickest solution to the selection problem, which was box-based selection. Command-select is used for selecting disjoint parts of the expression because that is how other desktop environments handle disjoint selection. It's awkward to use and not discoverable, but time kept marching on and I needed to have something implemented. With more time I will attempt to implement a cursor-based selection instead of box-based selection, so that it feels more like selecting text. I haven't yet thought up an alternative to using command for disjoint selection.

Shoreline was written to run on macOS using Swift. The alternative was writing it using either a language that compiled to JavaScript or JavaScript itself. I've spent too much time writing JavaScript to trap myself into writing any more. Targeting macOS heavily restricts the number of users Shoreline can potentially reach, but so do the many bugs in the application, so that's a secondary issue. I feel better knowing that Shoreline is simply a prototype from which I can take all the lessons I learned from writing it in Swift over to other languages and environments that might allow more people to use it.

The technical part of this project revolves around taking a tree representing a math expression, seeing if the tree "matches" a bunch of other trees, and replacing subtrees of the original tree according to the parts of the tree that have been matched. To the user, the tree that is being matched against is the expression that they are manipulating. Each node of the tree stores one or more children depending on it's arity. Atomic nodes (representing variables, numbers, etc.) don't have any children. Negative nodes (representing a unary minus) have one child. Division node have two children (numerator and denominator), and the addition and multiplication nodes have more than one child. Aside from how they appear to the user, these last two nodes are the same types. Nodes that I plan on adding in the future would correspond to taking the nth root of a number, raising a number to the nth power, and subtracting one number from another.

Selecting a part of the expression corresponds to marking one of the expression's subtrees. The first challenge I dealt with concerns how exactly to select expressions. For my first attempt, I simply gave each node in the tree a "selection index", which was a number representing if the node was selected, and with and what color the node was to be selected with. This worked great until I added trees with variable arity. Using selection indices and associating them with each node meant that I could either select each term in a multiplication expression individually or every term in the multiplication expression at once. My ill-thought out solution was to have the parent of each node store which children are selected using "selection ranges", which map selection indices to ranges of the node's children. This required having a kind of node that could only be a parent and had to be present at the root of every expression.



Selection Ranges:

red: (0, 1)
blue: (2, 3)

An addition tree with example selection

The first thing I implemented in Shoreline was the expression renderer, but I had to go back in and change how certain parts of the expression were drawn after nailing down selection. The algorithm for drawing each expression doesn't stray far from the basic algorithms that are taught in Fundamental of Computer Science I. I had the choice of drawing the expression to a canvas or using NSView elements for each part of the expression. I chose the latter approach because I needed access to the bounds of each subexpression in order to implement selection correctly.

Expression selection from a user perspective is slightly more complicated than it needed to be because of my use of selection ranges. Whenever the mouse is being dragged, I calculate the atomic views which are being intersected. I then mark the lowest common ancestor of the intersected views as selected, unless the lowest common ancestor corresponds to multiplication or addition, in which case all bets are off and selection ranges need to be calculated. This algorithm is complicated further by the inclusion of disjoint selection, which currently has bugs related to overlapping selection ranges and the logic behind command-clicking. None of the bugs in this area of the code are difficult to deal with, they're just numerous and related to UI, and therefore frustrating to debug.

Implementing rules required "pattern" nodes, which show up as colored boxes to the user. Pattern nodes are similar to generic types in a language like Java. They are able to match with any selected node, but once they've matched with one, they can only match with selected nodes equivalent to the first one they matched with. A rule is made up of two expressions that may or may not contain pattern nodes. My idea for an algorithm that would match a selected expression against a rule would take in a rule and the lowest common ancestor of a selected expression. As long as one side of the rule matched with the selected expression, the rule matches the expression. The algorithm for transforming the selected expression would be similar but would take the nodes that the patterns matched with and substitute them into the unmatched side of the rule. The tree resulting from this would then be placed at the least common ancestor of the selected nodes of the original tree. I have not worked this algorithm out at all so there's a good chance none of it works.

Thanks to time constraints and a general fear of failure, I decided to go with a much hackier approach to implementing rules. For each rule I spent anywhere between twenty minutes to two days hand-coding an algorithm that only worked for the given rule. My rationalization for this change in approach was that it gave me more control over the behavior of each rule than I'd have if I tried implementing a general solution. In retrospect I think I was just scared that the general approach wouldn't work and would leave me with a buggy expression renderer to turn in at the end of semester. Instead I went with the specific approach and have a buggy expression renderer that lets the user do basic algebra if they ask it nicely.

A general algorithm for matching and applying rules would have made the rules easier to debug. In Shoreline's current state of affairs, whenever a bug is found I have to dig through hundreds of lines of code, trying to remember what logic led me to write them in the first place, just to find the line of code where I assigned a node a new parent instead of copying the node. There are also plenty of bugs that are due to poor logic on my end. There are a lot of cases involved in "moving" a term into the denominator of the other side of an equation and I know I missed a number of them because there are still bugs in that rule. The bugs that are left over are difficult enough to find that I don't feel bad about leaving them in there, but I know one day I will be demoing the application to someone and the bugs will show themselves. I know this will happen because it already did for my user evaluations.

5. Evaluation

Only two of my math-haters were available to test out Shoreline, but I found someone who hadn't done math in a while to take their place. As with Algebra Touch's evaluation, I am friends with all of the testers, and was hoping that Shoreline would snuff out any ill-will they had toward me for making them do math. I was wrong. I may have made it worse.

Each of the testers were tortured through a gauntlet of five expression manipulation problems. The expressions were already entered into the application, but I had to given them a goal for each one. The problems were as follows:

1) $b = a$ to $a = b$,
2) $3 + 1 + 4 + 2 + 5$ to $1 + 2 + 3 + 4 + 5$,
3) $2 = (-1) + 3$ to $1 + 2 = 3$,
4) $\frac{a}{c} = \frac{b}{d}$ to $a * b = c * d$
5) $y = m * x + b$ to $x = \frac{y+(-b)}{m}$

These problems were chosen because their solutions require the use of the main five rules.

Before starting with each of the math-haters, I briefly showed them how to select expressions and told them what each view was for. Shoreline's state as a prototype necessitated this instruction, as a full version would include some way of showing users the basic gestures necessary to operate the application before they start using it. I took notes as the testers worked through their assigned problems and informally interviewed them about their experience when it was over.

All the math-haters solved the first problem with relative ease after attempting to swap the a and the b by selecting the entire equation. They all tried to select the entire $a = b$ equation at first, but then remembered that they had to select the expressions separately after none of the rules became available to them.

The second problem, which had the user sort terms of a sum in increasing order, caused users problems because of a bug that I may or may not have introduced the day of the user evaluation. Sometimes selecting multiple adjacent terms would fail to register in the transformations view, leaving the wrong rules highlighted. The testers also expected the ability to switch the position of terms that weren't adjacent to each other. This wasn't possible, and I'm on the fence about whether it should be. I had to show all of them how to solve this problem.

In the third problem, two of the three testers chose the rule corresponding to $a = -(-a)$ when trying to move the (-1) to the opposite side of the equation. This turned the equation into the form $2 = -\big(-(-1)\big) + 3$. I'm happy I put in the undo button, as I didn't trust any of the math haters to know how to get out of this form without help. Two out of three of the testers had to be shown how to solve this problem.

Problem four appeared to turn the math-haters into math-super-haters, and none of them were able to solve it without some assistance. One of the testers kept selecting $\frac{a}{c}$, pressing the correct rule with the wrong selection, ending up with $1 = \frac{b}{\frac{a}{c} * d}$, pressing undo, and doing it all over again. Shoreline's expression renderer isn't as elegant as Microsoft Word's, so the testers seemed to panic whenever they entered states involving fractions within fractions.

The final problem combined all the rules the math-haters were supposed to have come across in the first four problems. Two of the three testers were able to solve it correctly after some assistance. The other one gave up after seeing the initial expression.

The post-evaluation informal interviews were more hopeful than the evaluations had been. None of the testers were good about verbalizing their thought process, despite being reminded to, so I only got a glimpse of the thoughts behind their actions after the actions occurred.

One of the testers said that she knew the answer to each of the problems but was frustrated by her their inability to come to an answer. This could be explained by inadequate aspects of Shoreline's interface, or by the tester's lack of experience with it. The latter explanation is backed up by another tester, who said that they felt they understood how to use it only at the very end. Even in its current state, conducting a user evaluation that tests what it takes to be comfortable working with Shoreline could be worthwhile.

All of the users expressed confusion about the meaning behind the rules' icons. My excuse for their undeveloped nature is a lack of time put into developing what they look like. The icons are central to Shoreline's function and interface, so some sort of study looking at the effectiveness of different kinds of rules and rule icons would help greatly in Shoreline's development.

The current state of the expression renderer is atrocious. A common complaint was about the number of parenthesis that would sometimes be on screen at once. This complain might be related to the panic that set in whenever a nested fraction popped up in problem four. I didn't consider the intimidation factor that a bad expression renderer would bring to Shoreline until after the user evaluation. It's not a major problem for the prototype, but I'll soon have to dedicate time to making sure the expressions look good.

Expression selection was complained about a lot, and two of the three testers said they wished they could drag the terms around instead of selecting them. Dragging of terms won't be added to Shoreline because such a feature could break expandability and consistency, but I am interested in considering other methods of letting users match expressions to rules. However, there are other methods different than the ones in the prototype within the selection paradigm that would give users the same amount of power that the current method provides.

None of the testers mentioned the expression history, probably because its current state doesn't look or function like my original vision of it. Instead of being integrated into the main window it is hidden at the top of the screen in a small box. The user was supposed to be able to interact with the history in a way similar to but more powerful than they interact with previous work on paper. Implementing an expression history that the user is able to interact with would make the feature more important. Exploring an expression history's possible modes of interaction would be an interesting path to take going forward.

6. Conclusion

I want to work on Shoreline for as long as I can, and I hope I stay motivated and able enough to do so. Luckily, the prototype is nowhere near complete, so there is a lot of work to be done before it's anywhere near "finished". Assuming I have the time to do it, my first priority for Shoreline is rewriting the selection system and generalizing the way rules are applied. The next step requires that I better understand how users understand each rule.

Creating a short study that tests a person's ability to look at rules from the prototype and apply them to given algebra expression might help me choose effective rules to include in future versions of Shoreline. Interactive history and inter-expression manipulations are features that would greatly expand Shoreline's capabilities and would need to be evaluated. I still haven't performed a summative evaluation on Shoreline yet and hope to do one in the future.

It's a lot easier to dream about a project than to make it a reality, and without this directed study, Shoreline would still be an idea only realized in my notebook. Letting a project stay a dream is also less scary than forcing it into the real world. Projects I've set out to finish have always petered out in the past, taking with them some of the ambition I used to start them in the first place. These problems with past projects occur because of my mindset going into them as well with the development practices I used to make them real. I consider the work I did on Shoreline this last semester to be a success because of a change in mindset and a change in development practices. The change in mindset came from being able to speak to an HCI Professor every week about my project, and the change in development practices came from him forcing me into one that works. I intend to follow a version of the iterative design process that "forced" onto me for future projects because it set my expectations and priorities appropriately to ensure that Shoreline is properly realized.