Spring 2019

Visual Programming Language with Natural User Interface

Matthew Britton
mrb182@zips.uakron.edu

Table of Contents

Abstract

One of the fastest-growing fields of interest in computer science, fueled primarily by gaming, is the Natural User Interface (NUI). NUI encompasses technologies which would replace the typical mouse-and-keyboard approach to interaction with computer systems, with the goal of making human-computer interactions more similar to face-to-face interpersonal interactions. This is done by using technologies such as gesture recognition or speech recognition and speech synthesis, which use interpersonal skills we learn and practice on a daily basis. Visual Programming Languages (VPLs) are languages that allow the creation of a program by arranging graphical representations of program behavior, rather than textual program code. Visual programming tools are used in various disciplines, but are used most often for K-12 programming education, as a way to introduce fundamental programming concepts. This project is an application which combines these two ideas as an attempt to answer a question: Is it possible to do meaningful programming without actually touching a computer? The application uses the Leap Motion controller for gesture recognition, C# speech recognition functionality for speech recognition, and C# and WPF for the user interface design and logic.

Goals and Objectives

At the beginning of this year , my goal was to investigate what kind of approaches and tools might exist for creating accessible programs that meet the needs of individuals with disabilities or impairments, to facilitate common use of computers. To this end, I hoped to focus the project around computer accessibility by creating more tools that might aid in increasing accessibility. Unfortunately, interesting as these topics may be, I soon discovered that expanding on existing tools would be far beyond both the scope of an undergraduate project and my own capabilities. There are already many tools, both software and hardware, which attempt to improve computer accessibility in numerous different ways, many of which are tried, tested, and proven in their utility. Furthermore, these tools often have a large amount of support, both in manpower and financially, and so are able to be fully dedicated to their goals.

This is not to say that my time looking into accessibility was wasted, however. By looking at existing approaches, I was able to learn what has proven to be effective for accessibility and carry that information forward to new ideas. I found that there are several common approaches for accessibility for individuals who cannot effectively use a mouse and keyboard to provide input, or see a screen to receive output. A common output method is the use of screen readers, which are programs which can read and describe the content of a screen to the user. Screen readers exist both as standalone programs and utilities built into operating systems. For example, Windows provides the Narrator utility, or one can download any number of third-party screen readers, such as JAWS (Job Access With Speech) or NVDA (NonVisual Desktop Reader). For people who are not completely blind, there are also other approaches to making visual information clearer. Often, operating systems will provide the ability to increase the size of icons, text, and images across the whole system. Other common options include high-contrast or black-and-white screen display modes, where visual information on the screen is intended to be easier to differentiate and distinguish. There are also several methods for providing input. There exist specialized braille keyboards to allow people with visual impairments to read and input text

to a system by changing a tactile braille display. A more common method is speech recognition, by which a user can speak commands or text to be interpreted as input.

After concluding that implementing a meaningful addition to accessibility tools was unfortunately not a feasible project, given both my abilities and the scale and scope of what the project would have to be, I concluded that a topic that would be just as interesting, both personally and academically would be an exploration of visual programming with Natural User Interface (NUI) components. NUI refers to technologies which would serve to replace the mouse-and-keyboard-based interactions of present computing. This could encompass speech or gesture recognition, or even image comprehension and recognition [3]. The drive behind these methods is to make interaction with a computer system more analogous to standard interpersonal interactions, such that instructing a computer to perform a certain task could be as simple as asking the same of a person.

Visual programming refers to programming by manipulating graphical representations of programming elements, rather than textual ones. That is, rather than typing code into an editor, the user can drag and drop or otherwise move code elements in sequence to create a program. Many Visual Programming Languages (VPLs) already exist for purposes ranging from elementary school education to 3D modelling and animation to simulation and automation. However, their most common and interesting use is in programming education. VPLs are often successfully used for early programming education to teach programming concepts while avoiding losing students in the details of syntax [1]. I decided to investigate existing VPLs by looking into a very well known and established example, Scratch.

Scratch was developed by the MIT Media Lab for K-12 programming education [2]. It provides mechanisms for animations and text manipulation. Scratch provides the user with a set of programming elements to drag and drop into sequence and manipulate to create a program. At any given step in the program, the user can see what instruction is being executed while its effect

takes place. Scratch can be used for anything from raw computation to animations, so I was not aiming to replicate it. Rather, I was looking to learn from its approach.

In Scratch, each instruction is represented by a block. The color of each block indicates what kind of behavior it entails - flow control, arithmetic, visual animations, audio cues, or any other language feature. The text on the block describes exactly what kind of behavior it performs, and the shape of the block is meant to suggest how it should fit together.

The example below is a sample of a Scratch program from the front page of the website, a program which allows the user to compose and play back music on a musical staff. Observing this small sample of Scratch, its utility as an educational tool is immediately evident. Whereas a snippet of code in a typical textual programming language could easily be unintelligible, it is immediately obvious what each line of this program does. Moreover, the visual design of each block, in both shape and color, helps to indicate what kind of functionality it performs and how it

fits together with other blocks in the program.

As another practical example of the utility of visual programming tools, my original plan for this project was to build it using HTML, CSS, and JavaScript, but it is now in C# using WPF. This is because basic development in web frontend languages requires the developer to make all changes in a text editor, or to use some type of third-party application. In contrast, Microsoft Visual Studio supports graphical creation and editing of graphical Windows applications, built specifically with this type of development in mind. Rather than having to remember and textually manipulate every element of the display, the developer can click and drag elements from menus, position elements on the interface visually, and use familiar methods of interaction to build the display. This allows development to be faster and easier, while also facilitating development of more complex projects. Given this, I chose to change the implementation language of the project to WPF in C#, which has built-in support from Visual Studio.

Methodology

I decided to use my Fall 2018 Human-Computer Interactions term project as a way to experiment with my ideas, as a first pass to explore what works and what doesn't, and provide a base implementation to work from in the future. The project was composed of two parts - a C# WPF application which allows the Leap Motion controller to control the mouse cursor on the screen, and a webpage designed in HTML and JavaScript which presents a simple front-end designed to work with NUI technologies. The primary lesson I learned implementing this project is that for an NUI application, the "feel" of the application is very important, even moreso than with a normal GUI application. When using a mouse and keyboard to control an application, a user can have fairly precise control of their input, and make small adjustments easily. When using NUI devices for gestural or speech recognition, this is not so simple. Often this type of input is much coarser than mouse movements and cannot be easily undone. Moreover, speech recognition in its simplest form can only provide input, and cannot edit previous input. With these considerations, applications must be designed to be tolerant of a fair degree of error.

In the web interface, I learned several lessons about designing an interface when an uncertain degree of precision is involved. Every element of the interface has a distinct background color to distinguish it from its surroundings. Since the user input comes as large movements and does not have a precise degree of control, it is important to distinguish disparate elements from each other. Also, each element has more internal padding, which also helps to account for the user's lack of precision. If the user accidentally makes minute movements, this will not disrupt their intended interaction, since the cursor is still within the intended element.

That particular application is undeniably trivial. However, its general structure is very common in computing; that is, arranging elements in sequence and performing some kind of computation on each one. In particular, this is relevant to visually arranging and stepping through

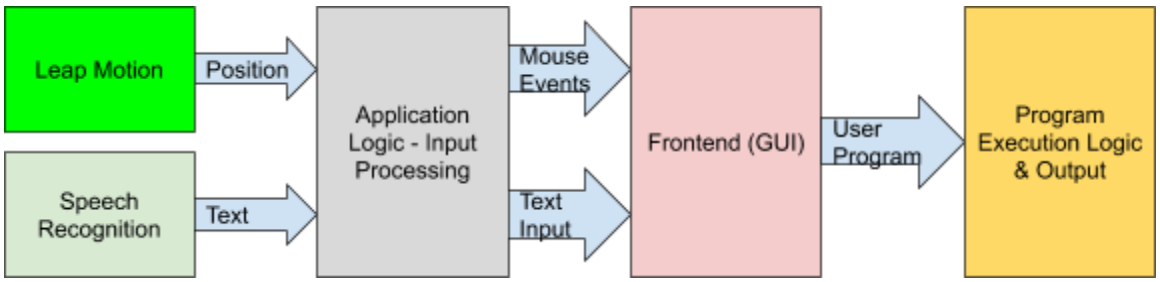
executable elements of a program. I was able to take the lessons I learned and experience gained by doing this term project and apply it to my honors project as a whole.

The first portion of the application designed was the interface. Based on existing examples, such as Scratch, I decided that each programming element should be represented by a visually distinct block, distinguished by color and text. As for the construction of the program, I decided that program elements would be dragged into a distinct area which would display them in sequence. I determined that program output would be provided through text, in a distinct area. Input during the construction of the program for gathering the necessary data for each statement would be performed by an independent dialog window. In all these areas, I attempted to minimize complexity as much as possible. This aids both in the “feel” of the application, as well as avoiding potential complexities and unintended or accidental interactions on the part of the user during the use of the application.

The second part which needed consideration was the behavior of the language itself. I decided that to demonstrate the utility of this application, it should at minimum be able to perform the basic functionality that all programming languages have in common: input, output, assignment, arithmetic, and some kind of flow control. Each of these statements would have to obey some sort of consistent internal logic to allow the application to best emulate the use of a typical programming language. The guiding idea that all behavior in the language follows is this: a program is a sequence of statements, each of which reads or modifies the state of a variable in the program. The only statements which do not follow this rule are loops, which read or modify the behavior of other statements. By laying out these fundamental rules, the design of the language maintains an internal consistency that simplifies both its development and its use.

The Leap Motion controller was first released in 2010, initially for creating new interactive games. It provides a mechanism to recognize a user’s hand motions and provide this data for programmatic use. I already had some experience programming using the Leap Motion

controller, so this was fairly simple to use in the project. I knew that I wanted to use Leap Motion to control the mouse cursor, so I simply needed to decide how in particular to do it. In the end, I decided that the application would use the location data of a recognized hand to move the mouse cursor, and the pinch gesture to simulate a left mouse button event. In doing this, I hope to emulate the feeling of picking up and placing down physical objects.



Language Design

In this visual programming language, there are two main kinds of language components: variables and statements. Much like any other traditional programming language, a variable contains a certain value to be used by the program, and a statement expresses some action that should be carried out by the program. By arranging statements and assigning and using variables, a user is able to assemble a program which performs a specific kind of activity.

Before discussing the details of each kind of statement and its behavior, it is important to make note of some of the limits imposed on the project by its nature, in comparison to a traditional programming language. First, and perhaps most importantly, in this visual programming language, there are no compound statements. Each statement exists as a single, self-contained block, which performs exactly one task. This stands in contrast to a traditional textual programming language, with a recursive statement grammar, which may allow for multiple programmatic operations to be performed in a single line or statement, such as in this valid line of C++:

```
int x = (0 < y && y < 10 ? 1 : -1) * (f(a - (b * c)) % d);
```

This single line is, at the highest level, a variable assignment; however, the recursive grammar of C++ statements means allows this to contain numerous behaviors: the conditional operator, relational operators, logical and, a function call, arithmetic operations within the arguments passed to the function, and so on. While this type of grammar allows for general-purpose languages to have great flexibility and the ability to execute a large amount of programmatic behavior in a single statement, this also necessarily increases their complexity. This type of complexity is certainly not inherently bad, but it does oppose the design philosophy that visual programming languages tend to follow; that simplicity and intuitivity should be maximized. This certainly applies to educational visual programming languages, as throwing an inexperienced student “in the deep end” may be counterproductive to learning. I have decided to follow similar ideas of simplicity and intuitivity for the design of my visual programming language, particularly given its NUI components. While NUI is meant to make interactions with a computer

system more intuitive and “natural”, as if to simulate interpersonal interactions, the current state of pure-NUI interfaces is not capable of expressing the same precision of input that, for example, a mouse-and-keyboard design could. Moreover, since the interface for my visual programming language is a GUI, it is inherently built for interaction via traditional interfaces. This, in order to avoid creating complexity that is, by nature, made for a different interface, I have decided that simplicity, regularity, and intuitivity are the ideal qualities for my visual programming language.

These principles of simplicity, regularity, and intuitivity are what has guided certain decisions I have made regarding the behavior of certain language features. For example, I chose to only provide one interface for arithmetic operations, despite the fact that there are two different types of variables which have different operations that are valid. Rather than having to decide between “Numeric addition”, “Numeric subtraction”, or any other type of arithmetic operation that exists in the language, the user needs only to select “Arithmetic”, and provide the necessary arguments to the operation. The application will determine what type of operation is being attempted, and whether it is valid. A necessary consequence of this is that the user is allowed to construct statements that are invalid before program execution. Though this may seem counterintuitive at first, this is also completely in line with a traditional textual programming language. No language feature stops a programmer from inputting garbage into a text editor; rather, this error is caught after the fact, during compilation or interpretation of the program.

Just like any other programming language, data is stored in variables. Each variable must have a type and a name, and no two variables may have the same name. Variable names are case sensitive. The acceptable range of values that a variable may contain is determined by its type, Number or String, according to the underlying implementation. A Number may contain any value representable by a C# double within their range of (+/-)5.0 x 10^-324 to (+/-)1.7 x 10^308, and a string may contain any character string representable by a C# string.

In this visual programming language, an executable action or instruction is represented by a Statement. All statements perform some action on the stored value of a variable. Statements reference variables by name, as input by the user. To reference a Variable by name,

the user must match the name of the variable exactly, as determined by C# string equality comparison.

A Declaration statement represents the declaration of a variable in the program. All variables require a name and a type, which are provided by the user when they add the Declaration statement to the program. Variables are of either String or Number type, and the user selects the type of the variable from a drop-down menu. A valid variable name is technically any valid C# string; however, just as in any other programming language, attempting to declare a variable of the same name more than once is invalid. A variable declaration also includes the initialization value of the variable. Once the variable has been created, the initialization value is assigned to the variable, assuming it is of the correct type.

An Assignment statement represents the assignment of a literal value to a variable. The user provides the name of the variable to assign the value to, and the literal value which will be assigned. Literal assignment values are required to be of the same type as the variable to which assignment is intended; however, the details of determining whether or not this requirement is met by any given assignment statement can be non-obvious. The user input for the name and assignment value of a variable are both gathered as C# strings, and this data remains a string until the execution stage of the program, when assignment is attempted. The application decides exactly what kind of assignment is intended examining the content of the data string containing the value to be assigned. If the data string is a floating-point number, as determined by C#'s Double.TryParse() logic, the program attempts to assign the variable as a Number. Otherwise, the program attempts to assign the variable as a String.

This behavior was one of two main options I faced when determining the behavior of an Assignment statement. The first behavior I attempted to implement used the programmatic type of the variable (i.e., whether the variable was of class NumberVar or StringVar) to decide what kind of assignment to perform, and whether a conversion was required. However, this behavior had certain consequences which I decided I did not want in the language. Primarily, this had the result that, if the user provided a Double-format string as the assignment value for *any* type of

assignment, that assignment would be valid. This would mean that if a user had accidentally declared a variable as a StringVar rather than a NumberVar, if they believed they were performing numeric assignment, they would actually be performing string assignment. Then, if the user attempted to continue to mistakenly use the StringVar as a NumberVar, it could lead to unintended results. This takes control from the user and gives it to invisible internal logic, which is something I wanted to avoid. By deciding what type of assignment to perform based on the format of the user input, the application should adhere more closely to the user's intended behavior.

An Arithmetic Operation (internally an ArithmeticOp) is a statement which represents the evaluation of an arithmetic operation on two variables and the assignment of the result of this operation to the value of another variable. An Arithmetic Operation must satisfy several conditions to be valid. All three variables provided as arguments of the operation must have been previously declared in the program as part of a Declaration statement. These variables can be any declared variable, meaning that an operation in which all three arguments are the same variable can be a valid operation. In order to be valid, the operation must also be defined between the types of the source operands. There are four different combinations of source operand type pairings:

NumberVar @ NumberVar

NumberVar @ StringVar

StringVar @ NumberVar

StringVar @ StringVar

For each of these pairings of operand types, the operator determines whether the operation is valid. Valid operations are:

NumberVar + NumberVar

NumberVar - NumberVar

StringVar + StringVar

All cross-type operations are invalid.

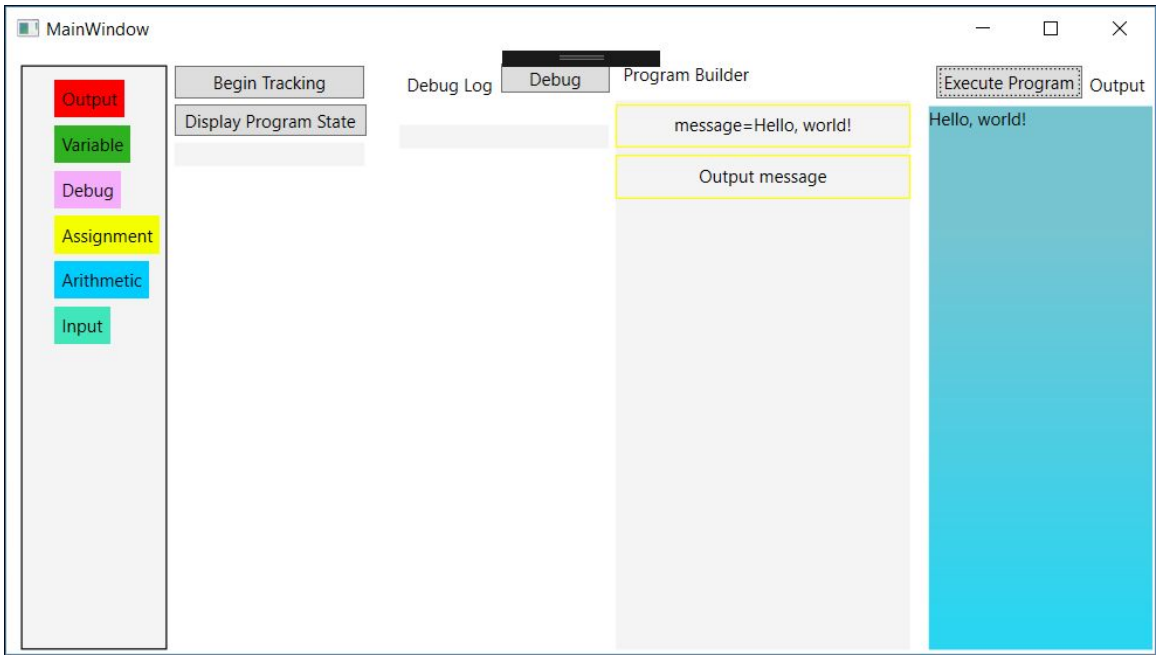
Most traditional programming languages also follow these same rules. However, this language differs slightly from the behavior of other languages by disallowing operations of the form StringVar * NumberVar and StringVar + NumberVar. Certain languages, such as Ruby, define multiplication between a string and a number to be string reduplication. For example, “foo” * 3 might result in the string “foofoofoo”. However, my language represents numbers as doubles rather than integers, and multiplying a string by .33 or 1.5 would logically require StringVar division to create fractional strings, which I have already decided is invalid. Similarly, many languages define an operation of the form string + number to perform concatenation of the number to the end of the string. For the sake of consistency, since no other cross-type operations are allowed, I have also decided that this kind of operation is invalid.

An Output statement represents the writing of the stored value of a variable to the application’s output area as text. A valid Output statement requires the variable provided to be previously declared in the program. An Output statement can output both StringVar and NumberVar variables. An Output statement always follows its output with a newline.

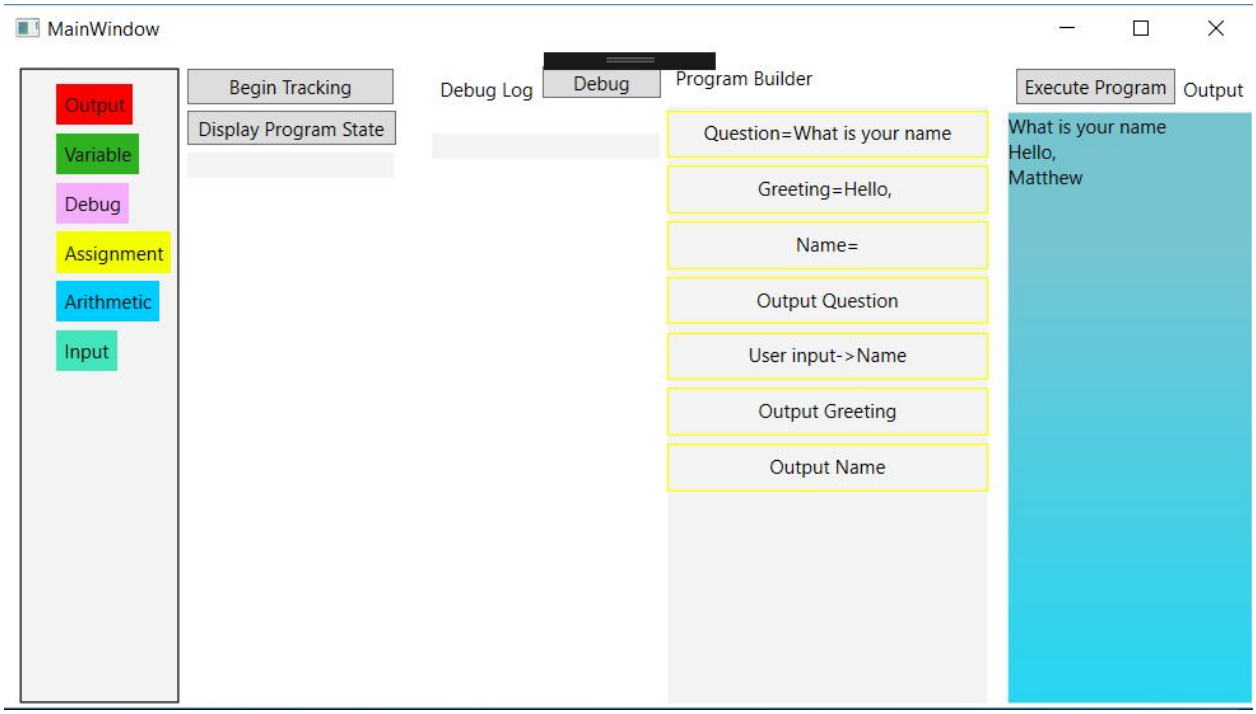
An Input statement represents the program in execution taking input from the user and storing it to a variable. As with all other statements, the variable provided to the Input statement must have been previously declared. Unlike an Assignment statement, for which the application attempts to perform the user’s *intended* type of assignment, the Input statement examines the type of the destination variable and uses that information to interpret the user input as the corresponding C# datatype (double or string). This leaves open the possibility that a user could provide invalid input to an Input statement, by providing a non-numeric string as input to a NumberVar. However, I felt that this was acceptable, given that other traditional strongly-typed languages can have similar complications.

Finally, a Loop statement represents the program repeating a block of other statements a given number of times. To introduce the concept of scope, or hierarchical ownership by a Loop of other types of statements, I introduced a new element: the End Loop element. This does not perform anything on its own, and does not exist as a Statement in the program. Rather, it is a

signal to the application to group a number of preceding statements into the body of a loop, at the time of program construction. A Loop must have a corresponding Loop End, and any Loop End must close an already opened Loop in order to be valid. Loop statements execute their body a constant number of times, as provided by the user.



The above image shows a simple “Hello, world” program to demonstrate the basic layout of the interface. At left, the distinctly colored labels each correspond to a type of statement that may be added to the program. This is done by dragging and dropping the label indicating the desired statement to the section labelled “Program Builder”. When a label is dropped into the Program Builder, after the required information is gathered from the user from a dialog window, it appears in the Program Builder with a distinct outline, containing text indicating what behavior that particular statement describes. To the far right,the teal box is the text output area, where Output statements write their output. Above this, the Execute Program button begins the execution of the current state of the program. Technically, the application can attempt to execute any program state, even an empty program, but only a valid program state will produce meaningful behavior.



This example demonstrates slightly more complex behavior with string input and output. This also demonstrates a significant difference from conventional programming languages - there is no such thing as a temporary or literal variable. In C++, for example, the behavior of this program could be reduced to

```
std::string name;
std::cout << "What is your name?\n";
std::getline(std::cin, name);
std::cout << "Hello, " << name;
```

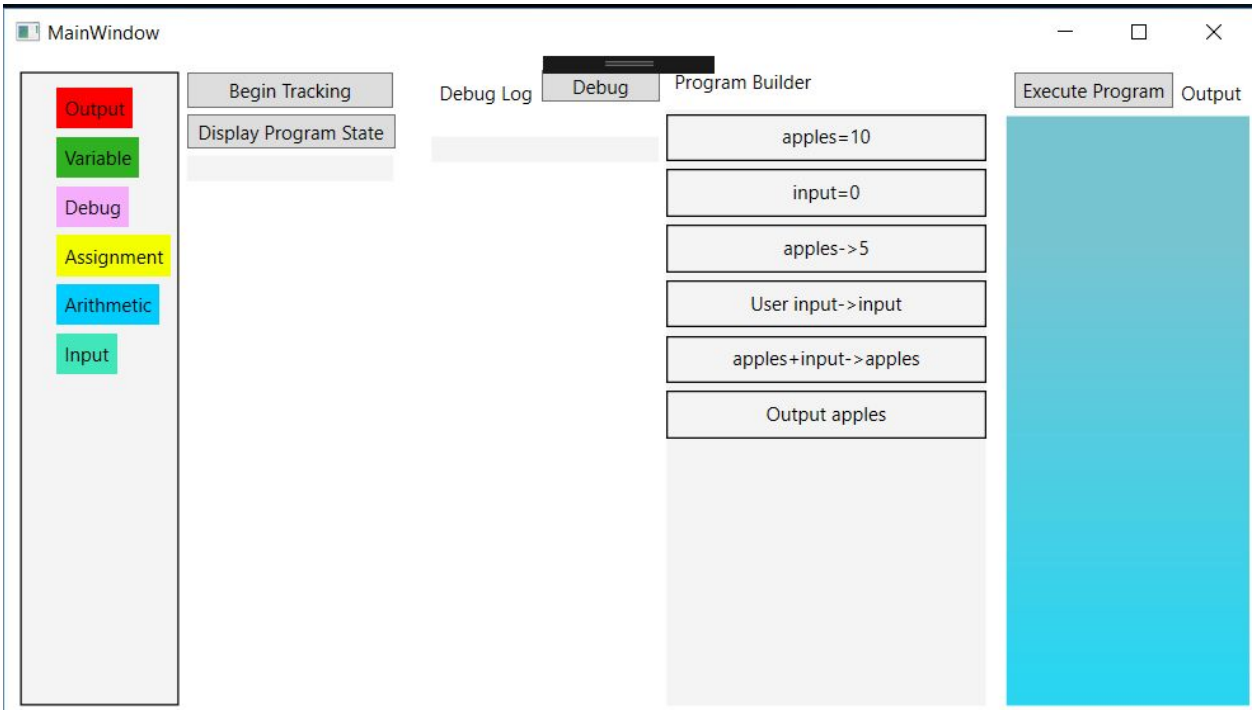
In this segment of code, the strings “What is your name?\n” and “Hello, ” are both string literals, meaning that the value is stored but is not referenced by a variable name. Behind the scenes, though, these values still must be stored in memory somewhere. The difference here is that in the visual programming language, the only way to use a value in a statement is to store it in an explicitly declared variable. Even for a single-use string message, the only way to display it is to store the message in a variable and reference it later in an Output statement, shown in this example by the following statements:

Question = What is your name

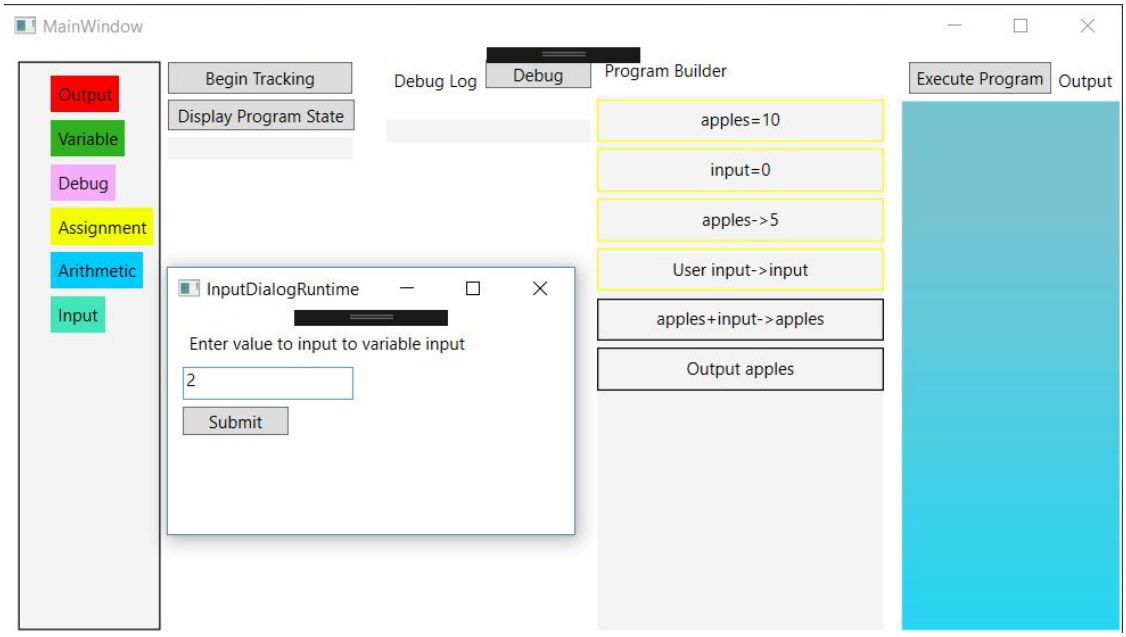
(...)

Output Question

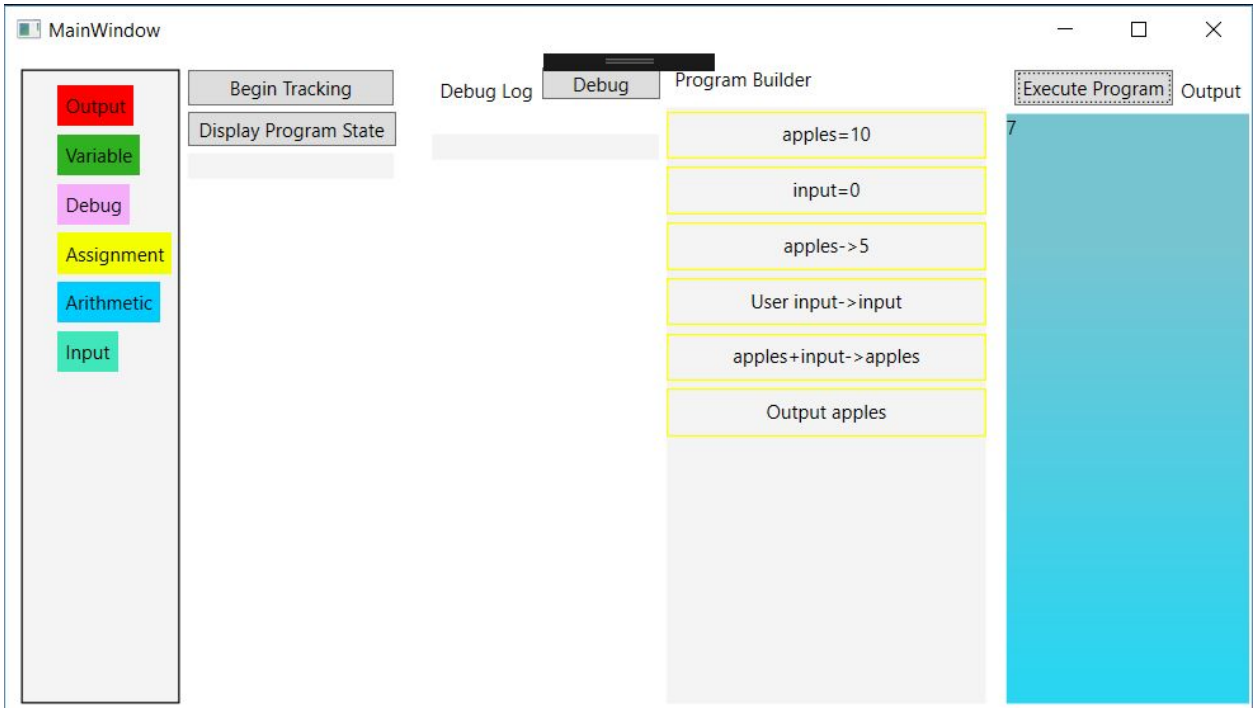
There is not a way in the language to create a statement of the form *Output "FooBar"*, for example.



This example shows numeric arithmetic, input and output. The program first declares two variables: apples, initialized to 10, and input, initialized to 0. Then it assigns 5 to the value of apples. It takes user input and stores it in the variable input. It adds the values of apples and input, and stores it to apples. Finally, it outputs the value of apples.



The image above shows an intermediate stage during the execution of the program. The runtime input dialog gathers input from the user and stores it to a declared variable.



This image shows the final outcome after the execution of the program. This demonstrates the correct output of the program based on the user input provided (i.e., 5+2=7).

This language is very high-level, and the program exists only in working memory, so its execution most closely resembles interpretation, such as with JavaScript or Ruby, rather than compilation. This also means that error checking only happens during the execution of the program. If a user creates an invalid statement, such as attempting to output an undeclared variable, this error will be detected during the execution of the program, and the execution of the user's program is terminated.

Implementation Details

The application is implemented in C# and XAML, an XML extension for designing WPF Windows Forms GUI applications.

The GUI frontend of the application consists of several fundamental parts. Since this application is drag-and-drop visual programming, there obviously must be something to drag and something to drop. In this case, the various kinds of “statements” or programming elements in the visual programming language are represented by several WPF Label controls, the textual content of which indicates what kind of statement it adds to the program. These are visually distinguished by their background color in addition to their textual content. These draggable elements are then designed to be dropped into a WPF Grid element. When an item is dropped into the grid, it will visually appear in this grid, reflecting the statement being added to the logical Program object in the UI. There are also several Buttons on the main UI, two of which are relevant for this discussion. The first, which contains the text “Begin Tracking”, is responsible for the construction of the Leap Motion Controller object, so that the program can track and use the gesture recognition data provided by the Leap Motion tracker. The second is labeled “Execute Program”. This causes the application to attempt to execute whatever the current state of the user’s Program is. The final one is a “Clear Program” button, which deletes the current state of any program the user is building. Any input is handled by the application when it comes across a statement that requires input from the user, and output is written to a TextBlock line by line.

During the construction of the Program, input is handled by custom dialog windows, corresponding to each specific type of statement. In total, there are seven types of dialog windows. Five of these are used during the construction of a program: ArithOpDialog, AssignmentDialog, DeclarationDialog, LoopDialog, InputDialog, and OutputDialog. Each of these dialogs is responsible for gathering the necessary information from the user for their respective statement type. For instance, the ArithOpDialog gathers four important pieces of data: the left- and right-hand operands to the arithmetic operator, the arithmetic operation to apply, and the

variable in which to store the result. Each Dialog window has two buttons: Submit and Cancel. If at any point, the Submit button is pressed, the dialog window will be closed and the main application will attempt to read whatever values are provided, whether they are valid or not. Alternatively, when the user clicks the Cancel button, the dialog window is closed, and the state of the Program does not change. The sixth type, InputDialogRuntime, is used during the execution of the program to gather input from the user. Unlike the other types of dialog windows, this one does not have a valid “Cancel” operation, since the user must provide some type of input to the program in order to continue valid execution.

Each of the program-construction phase dialog windows also contains speech recognition logic for certain input text fields. This is enabled by classes in the System.Speech library; specifically, the SpeechRecognitionEngine class. A SpechRecognitionEngine reads an audio input stream as input, and attempts to generate a textual transcription of speech in the input stream based on a provided grammar. This grammar is represented by the Grammar class, and can be either constructed programmatically to meet the needs of a specific application, or a default recognition grammar already on the system. This application makes use of both of these approaches for different purposes.

The primary function of speech recognition in this application is to recognize the names of variables said by the user, so that they do not need to use the keyboard to provide textual input. When the user adds a Declaration statement to the program, the DeclarationDialog window is responsible for gathering the name, type, and initial value of the variable to be declared. In this instance, the application attempts speech recognition when the user first clicks into the textbox which will contain the name of the variable, within the GotFocus event handler of the textbox. Here, the application uses the system’s spelling recognition grammar to gather the name of the variable. Originally, this used the default dictation grammar, however, this particular approach had some limitations due to the nature of the default recognition grammar. This grammar is the grammar used by the system-wide accessibility tools, so the

SpeechRecognitionEngine often attempts recognition of full phrases and natural sentences where these may not be the desired input. For instance, if the user speaks the word “name” during recognition, the engine may transcribe this input as “my name is”, “the name of”, or other similar phrases. While this has utility in a general use context, these phrases are certainly not ideal variable names.

After facing these challenges, later in development, I changed the recognition grammar to the system’s spelling grammar. This allows the user to spell a word or phrase letter by letter. Originally, to handle the aggressive phrase recognition, the application would only attempt this recognition the first time the user triggers the GotFocus event. Any following times, under the assumption that the user is returning to correct an undesired result, the application insead makes note of the change in a log file by appending a new line to the content of the file. This new line contains the time and date of the change, followed by the original text that was changed. This data provides the ability to analyze how many changes a user has made, and what kind of changes they were, providing insight into how to improve the application’s recognition behavior in the future. Following the change to the spelling grammar, it is often not necessary to use this feature, as the desired spelling is often correct the first time.

The other construction-phase dialogs also use speech recognition for variable names, but in a different manner. Since, in order for any other statement to be valid, it must reference a variable that has already been declared in a Declaration statement, the names of all potentially usable variables are available to use for recognition. This means the user is providing a choice from a list of possible valid options. C# speech recognition supports this specific pattern through the use of the Choices class. An instance of the Choices class simply represents a set of choices from which the user must choose, constructed from an array of strings. This instance of Choices can then be used to construct a GrammarBuilder, which is then used to create a Grammar for recognition. Following this pattern, the names of all previously declared variables are provided to an instance of Choices, which is then used to construct a simple Grammar, consisting of only

these variable names. Essentially, this means that any SpeechRecognitionEngine following this Grammar is selecting the variable name that it determines to be the best fit for the provided audio input.

C# through WPF provides very powerful functionality to support any number of drag-and-drop operations with highly customizable visual and programmatic effects. These are supported by several events belonging to the drag source and the drop destination Controls, facilitated by the static DragDrop class.

```
//Click and drag a UI program element
private void ClickAndDragElement(object sender, MouseEventArgs e)
{
    //A drag event consists of a MouseMove while the left mouse button is down
    Label outputLabel = sender as Label;
    if (outputLabel != null && e.LeftButton == MouseButtonState.Pressed)
    {
        //Call drag&drop functionality from DragDrop static class
        DragDrop.DoDragDrop(
            outputLabel, //The source of the data - typically source of event
            outputLabel.Content.ToString(), //Object containing transferred data
            DragDropEffects.Copy //Permitted DragDrop effects
            );

    }
}
```

The data source for the drag of the drag-and-drop operation first must have its MouseMove event handled. When the Control receives a MouseMove event while the left mouse button is down, this begins the process of “dragging” data from the element. From there, it is simple to initiate a drag operation using the DragDrop class. Its DoDragDrop method handles the encapsulation and transportation of data between controls, and its only programmatic requirement is the source of the data, the object whose data will be transferred, and an enumeration value from DragDropEffects which determines which drag-and-drop effects are permitted by the current operation. This application only uses the Copy effect to copy the string content of the UI control, but there are a number of other effects that are possible, such as Move,

which removes the data from the original control, or Scroll, which allows the window to scroll to the drop target while the drag operation is in progress.

To enable dropping into a Control, its AllowDrop property must have the value “true”, and there must be a handler assigned to its Drop event. In this application, the program builder UI element is a WPF Grid control whose Drop event handler is responsible for both updating the logical Program class, and its own appearance in the interface. The pattern of this Drop event handler is relatively straightforward. After reading the data from the event argument, it dispatches its behavior based on the content of the event, which determines specifically which type of Statement should be created. All Statements retrieve some kind of information from the user, which is then used to construct a new Statement of that type. This data is also used to modify the appearance of the program-builder Grid in the main interface. The final step of this event handler is to add the new Statement to the end of the Program’s list of Statements.

When the “Begin tracking” button is clicked, a Leap Controller object is instantiated and assigned an event handler to its FrameReady event. This handler is responsible for the tracking functionality of the application. If the Leap Motion controller only detects one hand in the frame, the app reads the Leap Motion X and Z coordinates of the palm of the hand and converts them to X and Y coordinates for the mouse pointer. The application reads the “pinch strength” of the hand, a value between 0 and 1. If the pinch strength is over a given threshold, 0.75 in this case, the application considers the left mouse button to be down; otherwise, the left mouse button is up.

In order to simulate virtual mouse actions, the application uses functionality included in the system library “user32.dll”. Specifically, it references two functions, SetCursorPos() and mouse_event().

mouse_event(uint dwFlags, uint dx, uint dy, uint cButtons, uintdwExtraInfo) is responsible for simulating left mouse button up and down events in the application. dwFlags accepts hexadecimal integer arguments from an enumeration which determine what type of mouse action

should be taken. 0x02 represents a left mouse button down event, and 0x04 represents a left mouse button up event. The arguments dx and dy are the pixel offset from the top left corner of the screen where the event should be fired. The other arguments, though they are relevant in other mouse events, are not relevant to simple left mouse down and left mouse up events, and so are not used in this application.

SetCursorPos(int x, int y) is fairly simple. Its parameters, X and Y, are integer pixel offset from the top left corner of the screen. Although it may seem obvious that these mouse coordinates should be at maximum 1920x1080, the typical pixel resolution of a monitor, this is in fact not exactly the most optimal configuration for this application.

To translate between Leap Motion and application coordinates, the following equations may be applied.

$$x_{app}=(x_{leap}-Leap_{start})App_{range}Leap_{range}+App_{start}$$

$$Leap_{range}=Leap_{end}-Leap_{start}Leaprange=Leapend-Leapstart$$

$$App_{range}=App_{end}-App_{start}$$

In this application, the relevant Leap Motion coordinates are $X,Z\epsilon[-200, 200]$. Translating these to screen coordinates might initially seem trivial, as a typical screen has coordinates 1920x1080. However, while this technically covers the whole screen, it makes reaching any extreme edges of the screen difficult, as the controller's tracking begins to deteriorate, and the controller does not provide high enough precision to capture minute differences between hand positions at these extreme edges. Therefore, the application maps to screen coordinates 2000x2000. This allows room for both user and device error and makes the whole experience more comfortable.

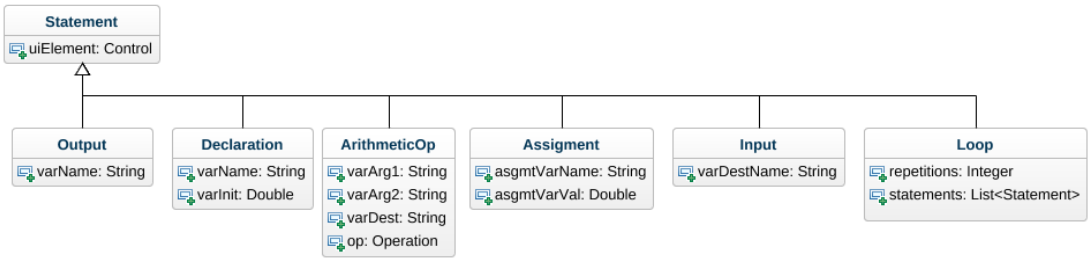
The structure of a program created by the user is represented logically as a class, Program. Each Program consists of two Lists – a List of Statements and a List of Variables. Statement and Variable are each base classes for a small class hierarchy which represents the

different types of Statements and Variables that exist in this Visual Programming Language. The Program class only functions as a container for the different elements of the programming language, and does not actually perform execution of the user's program or its statements. This is because WPF forms keep fairly strict control of what is able to access and reference their contained data, so accessing and using this data from an outside class is impractical at best, and impossible at worst. This also means that the execution logic for the Program class is contained within the logic of the main application window.

The execution logic is fairly simple at a high level. The application simply iterates through the Statements contained within the Program, and based on the dynamic type of each one, takes a certain route of execution. C# makes it quite simple to test the dynamic type of an object with the keyword *is*, and to reinterpret a base class reference as a derived class reference with the keyword *as*. For example, the code below tests whether the current Statement element has the dynamic type Output, and then performs computation on it by reinterpreting it as an Output:

```
//Outputs display the value of the contained message/variable
if(elt is Output)
{
    writeToOutput(elt as Output);
}
```

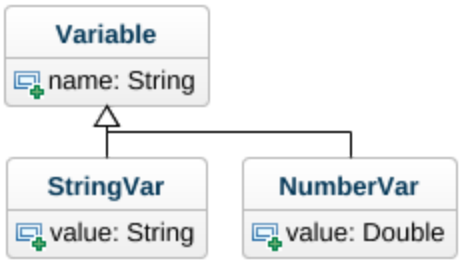
This logic similar for the other Statement subclasses. For instance, if the Statement has the dynamic type Assignment, the application casts *elt* as Assignment, then uses the data contained to assign the variable with the name contained in its asgmtVarName field to the value contained in its asgmtVarVal field.



The Statement hierarchy is the largest hierarchy in the application. The base class Statement contains only a reference to a Windows Form Control object. This provides a way to link a graphical UI element to its corresponding logical Statement. Currently, this only supports a small piece of functionality; that is, changing the outline color of a Control object to indicate that a statement has completed execution. However, in the future this could provide more ways to interact between the Program and the interface during the execution of a Statement.

The Statement class and subclasses represent any executable activity that the program can perform. There are several subclasses: Input, Output, Assignment, Declaration, ArithmeticOp, and Loop. Each of these contains their own additional information relevant to the processing that they represent. Output contains a string which denotes the name of the variable whose value will be written to output. Assignment contains both the name of the variable to assign to, and the value which it will assign. A Declaration contains the name of the variable to create, as well as the initial value. An ArithmeticOp is the most complex of the Statement subclasses, containing the names of the left- and right-hand operands of the operation, as well as the name of the variable which will store the result of the operation, and an enumeration value indicating which operation to perform.

When data for a particular type of Statement is gathered from a user, it is stored in a small container class whose only purpose is to encapsulate the data, in order to simplify passing it between different parts of the application logic.



The class Variable represents a variable declared by the user in the Program class. Whereas Statements are constructed and added to the Program prior to execution, Variables are created and added during the execution of the Program. The base Variable class contains only one field, a string containing the name of the Variable. Since the values stored in a Variable vary in type based on the type of the Variable (e.g, NumberVars encapsulate C# doubles, and StringVars will encapsulate C# strings), and there is no meaningful value for a Variable without a type, the base class does not contain any field for a value. This is left to the derived classes to implement. A NumberVar encapsulates a C# double as its value. Similarly, a StringVar encapsulates a C# string as its value.

Outcome and Results

In order to test my application to truly understand whether I achieved my goal, I knew that I needed someone other than myself to test it. I wanted to find someone young and relatively inexperienced in programming, to test the intuitiveness of the design. At the same time, I wanted to avoid someone who was completely new to programming, to avoid needing to explain very basic programming concepts, such as variables and datatypes, or input and output. Essentially, I wanted to find someone who could sit down in front of the application and figure it out on their own, and still give high-level feedback on the feel of the application and how it conformed to a basic concept of programming. Fortunately, I had ready access to someone who almost perfectly fit my needs - my own younger brother. He is 16 years old and in 11th grade, so he is around the age when people are typically first introduced to programming concepts. He has also taken a semester-long introductory course in programming, taught in Visual Basic, so he has a basic understanding of fundamental programming concepts. This means his experience and feedback is from the frame of reference of someone who has had experience creating simple programs, not far beyond the scope of programs that can be built in this application. Most importantly, I had ready access to his time and feedback.

The first feedback he gave, and the one that I was expecting the most, was that variable declaration speech recognition is “terrible”. The original speech recognition grammar for variable declaration used the default dictation grammar for Windows accessibility tools, which often attempts to create a natural phrase from the audio input it receives. This is often not ideal for variable names. For example, for one attempted input “Number”, the recognition engine provided “Numbers it has to” as output. In fact, he ended up manually changing every variable name that the recognition engine provided. This is an unfortunate result, but not unexpected. Creating my own speech recognition system would be far beyond the scope of this project, so I had to use an existing tool outside of its intended use case.

After changing this particular grammar to the spelling grammar, these difficulties were significantly reduced. Although the recognition is still occasionally incorrect or inaccurate, it is not so often that an entire phrase must be rewritten or replaced.

Speech recognition was also the source of another of my brother's complaints - that variable values did not also have speech recognition. This is also a consequence of the speech recognition grammar I used. Not only does the default grammar preferentially recognize phrases over words, but it recognizes numbers as their full spelled-out name, not numeric characters. That is, it could output "one hundred twenty three" instead of "123". I explored a number of ways of handling this. First, I considered creating a custom grammar of only numbers. However, this approach is poor at representing the entire range of values of numeric datatypes, as it requires a separate entry for each single value, creating an enormously large grammar that consumes a large amount of memory and slows performance. The other approach would be to accept all numeric input in its spelled out form, and then process the input and derive the numeric value. This approach is also not ideal, since even if the processing was perfect, the speech recognition would still be unreliable. Given these challenges, I decided that it would make more sense and be more reliable to require values to be input using the keyboard.

The other particular criticism he gave was that the gesture tracking using the Leap Motion controller was unreliable and spotty. I would attribute this partially to my own design, but also to the Leap controller. The accuracy of the controller's tracking data tends to deteriorate as it approaches the outer boundaries of its tracking range. It can also sometimes just simply be inaccurate, and momentarily detect gestures that are do not happen, recognize facial features as a hand, or other imporecisions. However, I am certain that there are ways that I could smooth out the gesture and hand tracking logic and improve the way in which I translate motions to mouse actions. Perhaps a lower pinch threshold for firing a virtual left mouse button could make the application less likely to release a click-and-drag when the user does not intend it. However, any changes would come with a tradeoff. If I were to reduce this threshold, then the user might accidentally fire more click events in places they did not intend. Currently, the application only

tracks one hand, and stops tracking when it detects two hands. I chose to do this to reflect the experience of using a mouse with one hand, but there are certainly other ways to go about this. I could have tracked two hands, one controlling the position of the mouse cursor and the other controlling the firing of virtual mouse clicks. I could change the ranges of values both in the tracking range of the controller and the range of pixel locations they are mapped to. Perhaps there is an entirely different approach I could have taken, but did not consider simply due to my familiarity with mouse and keyboard interactions and how so many applications are built with mouse and keyboard as a default assumption.

During the times I observed my younger brother using the application, I also witnessed him learning and reinforcing programming concepts. For instance, several times he attempted to write the result of an arithmetic operation to a variable he had not declared, but when the application indicated that this was invalid, he learned to declare variables before using them. Over the course of writing several programs, he also began to declare variables in bulk at the start of the program, rather than immediately before their use, which I found to be interesting. I suspect that this has to do with the lack of visual cues (aside from text) differentiating statement types at a glance. He simply reported that “it was easier and made more sense” to declare variables all together at the beginning of the program.

Academic Impact and Lessons Learned

Completing this project served a three-fold purpose as a capstone to my computer science education. It served as an object lesson demonstrating many of the abstract practices and ideas discussed in Computer Science courses. It broadened my own exposure to ideas and challenges in traditional and non-traditional computing. It also had the benefit of exposing me to technologies that I would likely not have experienced in my education otherwise.

As with many programs and projects, the final implementation of the project is built on the lessons learned from previous failed attempts. My first attempt at implementing this project was completely backwards from how it ended in the final version. Where the current implementation treats statements and variables as representations with no logic, my original idea was to implement each the execution logic of each statement type as a member of the class. Then, the execution of the program would be a simple loop through all the statements in the program, invoking the execution behavior of each one. This would look like

```
foreach(Statement s in program) {
      s.Execute();
}
```

This idea is very attractive, but was not realistic. In Computer Science, a basic goal for any program is the separation of concerns. Each part of a program should only interact with data and classes which concern its specific functionality within the application as a whole. However, to support the desire for a simple execution loop, each class would necessarily reference the state of the greater program as a whole. How could a Statement know whether it was valid without a reference to the Program containing it? How can a Variable know if its name has already been declared without processing every other Variable in the Program? The reality is that they cannot and should not. If, in C++, a programmer writes the line:

```
int a = b + c;
```

What knowledge should a, b, and c have? What information should = have access to? These questions make no sense. A program is a sequence of instructions that change the state

of variables in memory. The only entity that should have a view of the program as a whole is the program responsible for its correct execution. Whether this is a compiler or an interpreter, only it and the programmer are responsible for the correctness of the program, not the statements it processes. A statement does not execute itself, it is executed by the machine running the program. For all of these reasons, I had to abandon my original design. By making Variables and Statements behaviorless containers and implementing execution logic within the Program class itself, I made the application design more closely reflect reality while also better following the ideal of separation of concerns.

I also both wanted and needed my project to be open to extensibility. When completing a single project or assignment for a class, typically this is not even a concern. Very rarely will anyone intend to return to an old assignment to do work on it. However, I had to be very conscious of designing this project to allow addition of new features. Fortunately, by sufficiently obeying separation of concerns, this also led to a program design that allowed itself to be open to extension. Since unrelated parts of the program reference each other as little as possible, any new additions also only have to concern themselves, and not any interactions with other behavior.

This project also demonstrated the necessity of design before implementation. When completing an assignment for a class, often it has a fairly simple or well-defined goal, so it is easy to immediately begin writing code and make decisions on the fly. However, with this project, there was no way I could follow the same “just write code until it’s done” approach. For each feature of the application, I had to consider exactly how I wanted it to behave, how it would fit into the existing program logic and interface, and how it would affect the future course of the program. An example of a feature that I failed to fully design before implementation was speech recognition. I had the general idea, and implemented it in one location in the application. Then, when I needed to implement it in other areas, I copied and pasted practically all of the code into several different locations. This is undeniably a signal of poor design, and I paid for my lack of consideration when making changes to speech recognition as a whole required making the same changes to the

same code across several different places. However, this also helped to demonstrate how these kinds of poor design decisions can happen. I knew what would have been good, but due to deadlines and goals, I neglected to take the time it would take to improve the design.

This project broadened the scope of my knowledge and understanding with regard to nontraditional human computer interactions. Both in my early investigations and my later implementations I encountered a number of concepts I had never encountered or considered, even in my Human-Computer Interactions course. Support of computing for the visually impaired was never discussed in coursework. Neither was speech recognition or speech synthesis for any reason, be they for supporting non traditional computer use, or even just out of academic interest. This has also made me aware of the challenges which one must face when attempting to create an application explicitly for non-traditional interfaces within a typical computing system. Even though the project uses speech and gesture recognition, at its most basic level it is still a GUI application. Much of my work focused on translating different forms of input to analogous mouse-and-keyboard interactions. Even with all this, the project still relies entirely on typical graphical output. There is much more that could be done with an application to distance it even further from conventional modes of interaction, but for this project's scope, it serves as a proof-of-concept.

At the beginning of this year, I asked a question: Is it possible to write a program without even touching a computer? What would this kind of system looked like? To find an answer, I designed an application supporting programming in a visual programming language, which uses gesture recognition for positional input and speech recognition for text input. Within the limits of technologies I can work with, my project has demonstrated that this is possible.

There are still many ways that the functionality of this application could be grown in the future. The most obvious way would be the implementation of further language features. Conditional evaluation and function calls would be the first, most important features to support more useful programming. Following this, expanding on the datatypes in the language by implementing variable types like arrays or lists would allow the storage of multiple values of the

same type for ease of access and retrieval. In terms of the user interface, the first immediate goal would be finding a way to meaningfully implement speech recognition for all areas of text input. Currently it is limited by the way the default recognition grammars behave, but it may be possible to create a grammar which can be used to distinguish different forms of input (i.e., textual or numeric) and process them accordingly. If this is not successful, there may also be a way of using different hand gestures and positions to indicate the type of recognition behavior that should be performed. Creating these features would allow the application to truly support hands-free programming.

References

[1] K. Powers, P. Gross, S. Cooper, M. Mcnally, K. J. Goldman, V. Proulx, and M. Carlisle, "Tools for teaching introductory programming," *ACM SIGCSE Bulletin*, vol. 38, no. 1, p. 560, 2006.

[2] M. Resnick, B. Silverman, Y. Kafai, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, and J. Silver, "Scratch," *Communications of the ACM*, vol. 52, no. 11, p. 60, 2009.

[3] W. Liu, "Natural user interface- next mainstream product user interface," *2010 IEEE 11th International Conference on Computer-Aided Industrial Design & Conceptual Design 1*, 2010.