GestDefLS: A GESTURE DEFINITION LANGUAGE IN SWIFT

by

Robert William Writtenberry

April, 2015

Director of Thesis: Dr. Mark Hills

Major Department: Computer Science

The application programming interfaces supplied by Apple for developing applications in the Swift programming language on iOS devices provide limited support when it comes to declaring gesture recognizers outside of those simple ones currently provided. GestDefLS seeks to provide the service of allowing the developer to define custom, single- or multi-touch gesture recognizers. The language has the benefits of providing a concise, easy-to-understand language for declaring gestures, in addition to being readily compatible within Apple's Swift programming language. Furthermore, the language provides a higher level of modularity in terms of separating gesture recognition code from code pertaining to what the application should actually accomplish.

GestDefLS: A GESTURE DEFINITION LANGUAGE IN SWIFT

A Thesis

Presented To the Faculty of the Department of Computer Science

East Carolina University

In Partial Fulfillment of the Requirements for the Degree

Master of Science in Computer Science

by

Robert William Writtenberry

April, 2015

Director of Thesis

Dr. Mark Hills

GestDefLS: A GESTURE DEFINITION LANGUAGE IN SWIFT

by

Robert William Writtenberry

APPROVED BY:

DIRECTOR OF THESIS: ______________________________
Mark Hills, PhD

COMMITTEE MEMBER: ______________________________
Nassehzadeh Tabrizi, PhD

COMMITTEE MEMBER: ______________________________
Qin Ding, PhD

CHAIR OF THE DEPARTMENT
OF COMPUTER SCIENCE: ______________________________
Karl Abrahamson, PhD

DEAN OF THE
GRADUATE SCHOOL: ______________________________
Paul J. Gemperline, PhD

Acknowledgements

I thank the members of my defense committee, Dr. Qin Ding, Dr. Nassehzadeh Tabrizi, and Dr. Mark Hills, for taking the time to evaluate my work and for the knowledge I have gained throughout my coursework in their classes to facilitate the successful completion of this thesis.

I would like to thank my thesis director, Dr. Mark Hills, for his guidance, knowledge, and support throughout the entire process of creating this thesis.

I would like to thank my girlfriend for her love, patience, and support during this extremely busy time in my life.

Finally, I would like to thank my parents and sister for always believing in me and encouraging me. Without their love and support, none of this would have been possible.

Table of Contents

List of Figures

Chapter 1. Introduction

Touch-based user interfaces have become commonplace among computing devices, especially within mobile phones and tablets. The different types and combinations of touches that the devices can detect and interpret are called *gestures*. Because gestures play such a critical role in the functionality of the applications that run on these devices, it is important for the developer to be able to efficiently and effectively write code that identifies the correct gestures that should be recognized. The Apple APIs contain functionality to produce a handful of commonly used gestures and variations on them. The developer must therefore define more complex gestures manually. For the purpose of this paper, a *complex gesture* is defined as a gesture that is composed of one or more variations of the gestures currently recognizable by classes already contained within the *UIGestureRecognizer* API, which is the library provided by Apple for the developer to use to define a gesture that should be recognizable on the user interface. A *gesture recognizer*, therefore, is a class that contains the code to process the type of gesture that the developer wishes to detect within the application. Gesture recognizers are typically declared for use, and mutated, in a view controller file. However, the actual code to create a recognizer for a complex gesture is written in a separate subclass of the *UIGestureRecognizer* class. Furthermore, unless the developer desires to utilize the default values along with one of the provided gestures, he or she must add additional code to ensure that the standard gesture being recognized is based on the exact values, or *properties*, desired. These values are set after the developer declares the gesture recognizer in the view controller and include such attributes as swipe direction or tap count. To illustrate this concept of declaring a gesture recognizer, consider the following example. If the developer wanted to create a double swipe gesture, using the *UIGestureRecognizer* API, where the swipes are both to the left, the following gesture declaration in Swift would be needed. First, the developer would create a standard *UISwipeGestureRecognizer* and set the *direction* property of the gesture to *UISwipeGestureRecognizerDirection.Left*, in order to indicate that the swipe motion should be to the left. Then, in the method called to perform some action once the gesture is recognized, *handleGesture* in this case, the developer would have to increment some global counter *count* after the first swipe is detected. Once the counter reaches 2, a sound, stored in the *self.someNoise* variable would be played through the speakers of the device.

```
let recognizer = UISwipeGestureRecognizer(target: self, action:Selector("handleGesture:"))
recognizer.direction = UISwipeGestureRecognizerDirection.Left

func handleGesture(recognizer: UISwiperGestureRecognizer) {
    count++
    if count > 1 {
        self.someNoise.play()
        count = 0
    }
}
```

Figure 1. Swift code for double swipe gesture

As seen in this example, a dependency exists between code written to declare the gesture and the code written in the action function, because of the different locations of the counter and the gesture recognizer declaration. Combining such a dependency with any code that might be present in a subclassed *UIGestureRecognizer* class file, complex gesture definitions can quickly become hard to read and understand, making it difficult to abstract out what gestures an application should actually be looking for. Additionally, extending upon Apple's provided toolset also adds more lines of code to the developer's project. A gesture definition language is proposed in this paper to simplify the process of creating complex gestures, while reducing the negative aspects of creating such a gesture in application code. In the language, gestures are passed as objects to a method call within a class that extends the *UIGestureRecognizer* class. The following call declares the same swipe gesture shown in the figure above.

```
let recognizer = DSL(target: self, action: Selector("handleGesture:"))
        .doGesture(
                Swipe(direction: .Left),
                Swipe(direction: .Left)
        )

func handleGesture(recognizer: UISwiperGestureRecognizer) {
   self.someNoise.play()
}
```

Figure 2. GestDefLS call for double swipe gesture

This is more efficient in terms of modularity, because everything is declared in a single call and properties can be mutated through the use of named parameters. Declaration of the gesture is initially the same. However, a method, called *doGesture*, is called to indicate the gestures that should be performed. In this case, two Swipe gestures are declared with a required direction of *.Left* an enumeration value stored within the language. Meanwhile, the *handleGesture* function has been limited to a single line of code, the function telling the *someNoise* sound to be played on successful detection of the gesture. Furthermore, the global counter is eliminated in the GestDefLS call. This modularization of code provided by the language allows much better readability, which promotes a better understanding of what gesture should actually be recognized by the application. The tool is used in conjunction with Swift, Apple's newest programming language, to assist in the development of gestures within iOS applications. One goal was to be sure to include a majority of the standard functionality of the Apple *UIGestureRecognizer* API, in order to ensure that GestDefLS operates in a familiar way to the developer. While the goal is not to provide a complete alternative to the *UIGestureRecognizer* API, it is desired that GestDefLS can provide a simpler way of declaring one of the already-available gestures if the developer wants one. Elimination of lines of code, improved readability, and the ability to define the most common gestures of touch-based applications are the three main goals of the proposed tool. The first two goals are discussed in Chapter 5, with an explanation of common gestures being discussed in Chapters 4 & 5.

Research Contributions

This thesis presents two main research contributions. First, we define a new internal domain-specific language (DSL), GestDefLS, for defining gestures using Swift for the iOS platform. Second, we evaluate the effectiveness of this DSL by comparing GestDefLS code with code written natively in Swift, showing GestDefLS is more concise and easier to read.

Chapter 2. Related Work

Although GestDefLS is a DSL specifically for gesture detection on the iOS platform, other work has been done in the more general realms of mobile development, DSLs, and gesture detection. Fowler and Parsons have done extensive work in the area of DSLs, making a distinction between *internal* and *external* DSLs, and explaining the importance of *fluency* within an internal DSL (2011). External DSLs are defined as languages with a new syntax or syntax different from the native language for which the DSL will be used. Furthermore, a parser is created in the native language for analyzing the custom syntax. Meanwhile, internal DSLs are written in the native language and syntax for which the DSL will be used. In both internal and external DSLs, syntax is declared using vocabulary specific to the task being accomplished by the language. GestDefLS has been designed as an internal DSL. Fluency is described as making calls to the language work similarly to the process of writing out a sentence, with a *fluent interface* consisting of chaining together method calls that continuously return an object to have another method be called upon. Fluency, for ease of reading, was a goal within GestDefLS, even though a fluent interface could not yet be implemented.

DSLs for Mobile Development

The large numbers of mobile devices sold and applications downloaded to these devices are the main indicators of the growth of the mobile operating system into one of the premier platforms for software development (Le Goaer & Waltham, 2013). Because of the many varieties of mobile operating systems available, DSLs have been created to ease the process of writing applications that can run on multiple kinds of mobile operating systems. In addition to conducting an analysis of a number of these DSLs currently available, La Goaer & Waltham present their own such DSL, called XMOB. This DSL is a model-based language for declaring high-level functionality that is commonly present in all mobile applications, yielding skeletons of code in each of the desired languages to accomplish the task on a given platform. These templates are then individually checked and completed by the developer. This allows the application to feel native to the specific operating system it is running on. Steiner, Turlea, Culea, and Selinger also present a DSL for cross-platform development (2013). Their unnamed solution is a model-driven approach, as well. It utilizes the Xtext code generation tools within the Eclipse IDE to provide a DSL split up into multiple smaller languages that associate with each of the three parts of the model-view-controller architecture, along with one for declaring general aspects of the application, such as its entry point. Unlike XMOB, this DSL has an emphasis in applications that communicate with cloud technology. A third and final example of the use of a DSL for mobile development is the Gade4all platform for developing videogames on these devices (Nuñez-Valdez, Sanjuan, Garcia-Bustelo, Cueva-Lovelle, & Hernandez, 2013). The platform is comprised of a graphical editor, DSL, template, and transformation engine. The developer designs the game and features within the editor and describes their interactions with the DSL. Then, templates are created by the platform and transformed by the transformation engine into code for use natively on the platform of the developer's choice.

DSLs for Gesture Detection

Control flow within gesture-based interfaces varies greatly from that of traditional interfaces, where a single point of contact, or a button, for example, is touched by a single object, or a mouse pointer, for example, to indicate the next processing step. Hoste and Signer have indicated that, due to the intricacies of processing the data created by multiple sources of input to handle control flow within gesture-based interfaces, there is a need for domain-specific languages to simplify the process of declaring gestures (2014). Throughout their work, they present multiple criteria, which they claim should be aimed for when creating such languages, with the goal of reducing the accidental complexity of describing gestures. They describe accidental complexity as the challenges caused by available tools, rather than the actual complexity of the problem. With the recent rise in popularity of touch-based interfaces within mobile phones and tablets, solutions have previously been developed to assist in the construction and declaration of complex gestures. Three of these include Midas, GeForMt, and GestIT. Although providing frameworks for creating programming language-specific languages to declare gestures, none of these solutions offer an implementation in Swift. Therefore, GestDefLS eliminates the process of the developer having to implement one of the proposed solutions in the Swift language. However, a combination of principles from these solutions was considered when developing GestDefLS.

Midas

Midas, a rule-based gesture declaration language, features five qualities: modularization, composition, event categorization, GUI-event correlation, and temporal and spatial operators. Modularization, similar to the concept within the context of the organization of code and its respective duties, ensures that a single piece of a complex gesture is a simple, well-defined part that does not include any portion of a gesture that could be completed solitarily. Meanwhile, composition involves ensuring that any complex gesture can be composed of even the simplest of gestures available within the language(Scholliers, Lode, Signer, & De Meuter, 2011). This principle of composition was adopted during the creation of GestDefLS in this paper. For example, a *pinch* gesture, or two finger touches moving towards or away from each other, cannot be supplied in GestDefLS because of the nature of the method that allows simultaneous touches to be recognized on the user interface at the same time. It is made possible, however, to recognize a pinch gesture by composing one *pan* gesture, a finger touch moving in one specific direction on the interface, with another pan gesture moving in the opposite direction from the first pan gesture. Consisting of a three-tiered architecture, a rule or set of rules in Midas is/are created by the developer, which are compared to a set of facts. A rule is composed of a prerequisite and a consequence. When a prerequisite matches a fact, the consequence, or action, occurs. An implementation of the language for the java programming language, called *JMidas*, is also presented as a concrete example.

GeForMT

GeForMT is initially proposed as a formal model for gesture detection, followed by an implementation of the model in a later paper. The model is proposed as a utilization of semiotics, or a combination of syntactics, semantics, and pragmatics. A gesture is declared with a number of constraints, if desired, right in the initial declaration (Kammer, Wojdziak, Keck, Groh, & Taranko, 2010). This concept was adopted in GestDefLS. When declaring a gesture in GestDefLS, the developer should be able to identify any aspects of the gesture in their initial declaration of the gesture. Although fairly trivial to define after declaration, parameterized properties promote ease of reading and the isolation of gesture property information to a single call. One idea to be adopted in future iterations of GestDefLS would be the *pose* vs. *composition* concept that is utilized in GeForMT. A *pose* indicates that two fingers should be close enough together on the interface to perform the gesture, like in a two-finger swipe, for example. In GestDefLS, such a gesture is defined in the same way that a gesture exhibiting the GeForMT *composition* concept, where two touches are a distance away from each other on the interface, would be detected. The platform-independent implementation of the language begins by parsing a list of formal gesture definitions into the data model(Kammer, Henkens, Henzen, & Groh, 2013). User recognition input is then compared to the gestures in the data model. Any functionality that has been assigned for a certain gesture is then activated, pending a match between input and a gesture stored within the data model. In addition to predefined gestures, GeForMT also allows for gestures to be created and added dynamically, if the application allows the user to draw out a gesture on the interface, for example.

GestIT

Another declarative language proposed for multi-touch gesture definition is GestIT. For use with both touch-based and full-body based gesture recognition interfaces, the language is based on the concept of linking together *ground terms* through the use of *operators* (Spano, Cisternino, Paternò, & Fenu, 2013). *Ground terms* are explained to be the point where *features*, or certain parts of the body, begin to be tracked by the interface. An example of the operators is an asterisk to indicate that the gesture defined through a ground term should be recognized indefinitely. A *predicate* or series of *predicates* can be assigned to a ground term in order to identify certain properties that should be attained by the gesture in order for it to be considered successful. Within GestDefLS, the concept of operators used in GestIT has been applied to the class structure. Similar to the *parallel* operator provided in GestIT, indicating that two or more gestures should be recognized simultaneously, the *AtSameTime* class was created within GestDefLS to serve the same purpose of identifying two gestures that should be detected at the same time. In future iterations of GestDefLS, more classes can be created to represent the functionality of these operators. Providing even more customization options in this sense would increase the number of available gestures capable of being produced by GestDefLS.

Chapter 3. The Design Process

The design process involved an examination of complex gestures currently available within iOS applications, followed by the design of a call to GestDefLS and the design of the structure of the system behind the language.

Locating Interesting Gestures

To find interesting gestures that GestDefLS should support, a review of a combination of iOS applications and code was performed. The code review was important and most informative, due to the possibility of extracting the exact pieces of code to be condensed by the language. One application that was discovered was a simple open-source application written in Swift. The application, called *Monkey Pinch*, featured the images of a monkey and a banana on a single interface. The user can tap either image to make a chomping noise play through the speaker of the iOS device (Begbie, 2014). Likewise, a *tickle* gesture is available in the app that, when completed on either the monkey or banana, a laughing noise is played through the speaker of the device. This tickle gesture consists of the user moving their finger in a dragging motion first in one direction, then in the opposite direction from the first drag, then in the opposite direction from the second drag. In the case of any of these drags, they have to record a distance of at least *25.0 geometric points* across the image for the gesture to continue to be recognized by the application. Another open-source application that was found to have a custom gesture within it was called *Circular Knob Demo* (Erholm, 2014). Once again, this was a simple open-source application. However, this one was written in the Objective-C language. The app features a knob that rotates like a dial around the middle of the user interface. The custom gesture featured within the application is a one-finger rotation gesture, which allows the user to rotate the knob by using only one finger, rather than the traditional rotation gesture from Apple that requires two fingers. Due to the non-open source nature of Apple and its products, there was a limited amount of source material to search through, however. Therefore, it was necessary to perform a more general top-level analysis on a variety of iOS apps to decide which complex gestures have actually appeared in implementation. One example of an application believed to have a complex gesture implemented is called *Touchgrind*(Illusion Labs AB, 2008). This application is a game where the user is presented with a top-down view of a skateboard inside of a skate park. The user controls the skateboard by placing two fingers on the skateboard deck and maneuvering through the park, performing tricks to earn a high score. One trick, called a *flip*, is performed by the user first pressing two fingers on the board to accelerate it. Then, while one finger remains pressed on the back of the board, the finger pressed on the front of the board is dragged off of the board. The back finger is then immediately released to perform the trick. This gesture became the inspiration behind the *press-and-drag* gesture that is described later on in the paper. Once these resources were thoroughly covered, a list of complex gestures that we might like to appear in an application was created.

Application	**Gesture**
Monkey Pinch	*tickle*
Circular Knob Demo	*One-finger rotation*
Touchgrind	*Press-and-drag*

Figure 3. Complex gestures within applications

Designing the GestDefLS Call

With the compiled list of gestures, we began to imagine how these complex gestures could be written down in our DSL. Initially, the structure of a call to GestDefLS was planned to be in the form of a fluent API, by chaining method calls together and always returning the same object with each call.

Designing the System

After designing the structure of a call to GestDefLS, the complex gestures that were provided as example calls were coded within Swift as they would be coded without the use of the proposed DSL. Next, the preliminary structure of GestDefLS was designed and coded. This first iteration consisted of a single *DSL* class as a subclass of Apple's *UIGestureRecognizer* class. The class consisted of all the logic necessary to perform desired functionality of the language: the required overridden gesture detection functions, the method to facilitate simultaneous, multi-touch detection, as well as the specific gesture calls. This design, although feasible, was difficult to implement, due to time constraints. Therefore, it was necessary to come up with a new design, which consisted of a more object-oriented approach. Although some of the fluidity of the call to GestDefLS was sacrificed, the new design was more practical. It consisted of the *DSL* class, *Gesture* super-class, *Gesture* subclasses for specific gestures, and the *Gesture* subclass, called *AtSameTime*, to process simultaneous touch detection. Additionally, if variations on the simple gestures were desired, this new design could be extended, as needed.

Testing the System

As part of the Xcode IDE software, a simulator is provided to test applications made for Apple products. This simulator allows up to two touches at one time on the interface, with a limited range of movements that can be performed. Therefore, due to the critical importance of the touch-based interface to the purpose of creating a multi-touch gesture definition language, it was necessary to test on an actual device with such an interface. Furthermore, it was required that the testing device be registered with an Apple Developer account. An Apple iPhone 6 was the platform for testing most of the time, with the iOS simulator being used for gestures requiring only one touch or a limited range of motion.

Chapter 4. The Swift Programming Language

Because the target platform for the use of GestDefLS was the iOS mobile operating system, it was practical to choose a language that was created to run natively on the platform. When beginning the research, the Swift language had just been announced as the successor to Apple's Objective-C language. Swift was created to be a language that would be easier to use and more powerful than its predecessor (Apple Inc., 2014). Implicit typing is one example of this ease and power. A constant *x* and a variable *y* can be declared in the following way:

```
let x = 2
var y = "Hello World"
```

Figure 4. Swift code for declaring constants & variables

A constant is declared with the "let" keyword, with a variable being declared by the "var" keyword. Also seen in the above examples, the Swift language eliminates the need for a semicolon at the end of a single statement. A semicolon is required after each statement, however, if there are multiple statements on a single line. If the developer wants to explicitly define the type of a constant or variable, in case it could be interpreted multiple ways by the compiler, as is the case with a double or a float, it can be accomplished by placing a colon and the desired type name after the constant or variable name, like so:

```
let x: Float = 2
var y: String = "Hello World"
```

Figure 5. Swift code for declaring types

One special feature of the language is the "optional" type, as denoted by the question mark immediately following the type declaration, which is shown in this example:

```
Let x: Float? = nil
```

Figure 6. Swift code for declaring optional type

A typical variable cannot be assigned a value of *nil*. Also, Swift does not allow variables to go un-initialized within an object unless they are declared as being of “optional” type. Therefore, an optional type is used if either of these circumstances is desired. When using optional variables, constants, or objects, any method called on them evaluates to be of optional type with a value of nil if the value of the optional is nil. In order to get the non-optional value, an exclamation mark is used immediately following the optional variable or constant’s name:

```
var z = x!
```

Figure 7. Swift code for retrieving value within optional

However, if the optional variable computes to *nil*, an error will occur at runtime. Therefore, it is important to ensure that the optional has a value other than *nil* if trying to retrieve the value as a non-optional value. Optional values were used somewhat frequently within GestDefLS. Named parameters are another feature of Swift used to the advantage of creating improved readability among calls to GestDefLS. Finally, first-class functions and closures, which exist in Swift, were also taken advantage of in the creation of GestDefLS.

The *UIGestureRecognizer* API

Implementation of gestures within an iOS application involves utilizing the *UIGestureRecognizer* API, which is part of the *UIKit Framework*. A gesture recognizer is added to the view controller using the pattern shown in the following code segment:

```
let recognizer = UITapGestureRecognizer(target: self, action:Selector("handleTap:"))
recognizer.delegate = self
view.addGestureRecognizer(recognizer)
```

Figure 8. Swift code for creating a tap gesture recognizer

First, a gesture recognizer is declared within the view controller file. This involves declaring a *target* and *action* for the recognizer. In the example, *self*, or the current view controller, is set to be the target. The *action*, or function to run when the gesture has been successfully recognized, is the *handleTap* function. Then, the *delegate* for the recognizer is set, to the view controller in this example. A gesture recognizer's designated delegate serves the purpose of controlling the messages being received from the recognizer and how cases are handled when simultaneous gesture recognizers are declared for the same view (Apple Inc., 2013). Finally, the recognizer is added to the designated view using the *addGestureRecognizer* method of the view object. A *view* is a specific portion of the user interface represented by an image or graphic. The set of tools used to create a gesture recognizer is comprised of classes used to recognize six main gestures typically found in applications built for a touch-based interface. These gestures include long press, pan, pinch, rotate, swipe, and tap gestures. The class for the pan gesture recognizer has a child class that recognizes a gesture called a screen edge pan. Each of the classes containing the code to recognize the specific gesture contains properties that can be set to customize the requirements for each gesture to be recognized. Some properties have default values, while others have none. Furthermore, some of the properties can only be accessed and not mutated. Apple has divided these gestures into two classifications: *discrete* and *continuous*. A discrete gesture is identified as one that is detected, when completed, and responds to this gesture once (Apple Inc., 2013). These include tap and swipe gestures. A continuous gesture is described as one that continues to send response information about touch location, for example, in order to allow for multiple changes to occur to the view that the recognizer is attached to throughout the gesture (Apple Inc., 2013). These include pinch, pan, rotate, and press gestures. The following figures represent each of the six basic gestures. A circle represents the touch of a finger on the user interface. A circle inside of another circle represents any touch in a single location on the interface lasting longer than a traditional *tap* gesture, or a *press* gesture. An arrow represents the movement of a touch from one position to another.

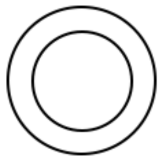

Figure 9. Gesture detected by *UILongPressGestureRecognizer*

The *UILongPressGestureRecognizer* class provides the ability to detect a *press* gesture. The properties that can be set within the class include the *minimumPressDuration*, having a default value of *0.5 seconds*, *numberOfTouchesRequired*, having a default value of *1*, *numberOfTapsRequired*, having a default value of *0*, and *allowableMovement*, having a default value of *10 points*.

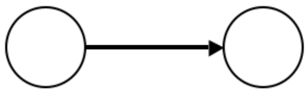

Figure 10. Gesture detected by *UIPanGestureRecognizer*

The *UIPanGestureRecognizer* class provides the ability to detect the movement of one or more touches from one specific location to another. The properties that can be set within the class include the *maximumNumberOfTouches* and *minimumNumberOfTouches* properties. The *minimumNumberOfTouches* property has a default value of *1*, while the *maximumNumberOfTouches* has a default value of *NSUIntegerMax*, the largest unsigned integer. The *UIScreenEdgePanGestureRecognizer*, the subclass of *UIPanGestureRecognizer* detect when a finger begins touching the edge of the interface and moves away from that edge on the interface. The property that can be set within the class is the *edges* property.

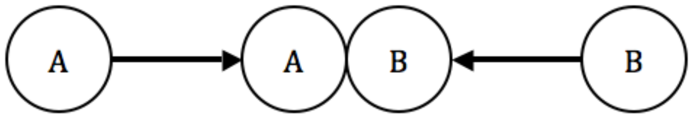

Figure 11. Gesture detected by *UIPinchGestureRecognizer*

The *UIPinchGestureRecognizer* class provides the ability to detect when two fingers are placed on the interface at the same time, and they are simultaneously dragged towards or away from each other. The property that can be set within the class includes the *scale* property. The class also has a property called *velocity*, which is read-only, and is affected by the value of *scale*.

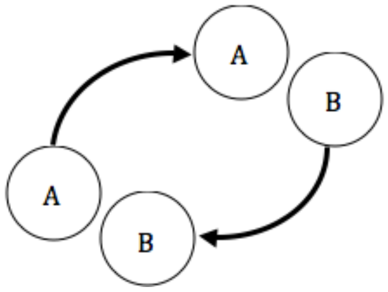

Figure 12. Gesture detected by *UIRotationGestureRecognizer*

The *UIRotationGestureRecognizer* class provides the ability to detect when two fingers are placed on the interface at the same time and simultaneously dragged in the opposite direction from each other, following some elliptical motion, in order to rotate the view for which the gesture recognizer is applied. The property that can be set within the class is the *rotation* property. The class also has a property called *velocity*, which is read-only, and is affected by the value of *rotation*.

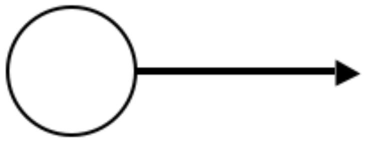

Figure 13. Gesture detected by *UISwipeGestureRecognizer*

The *UISwipeGestureRecognizer* class provides the ability to detect when one or more fingers touch the specified view in the interface and move in a specific direction. The properties that can be set within the class include *direction*, which has a default direction of *UISwipeGestureRecognizerDirectionRight,* and *numberOfTouchesRequired*, which has a default value of *1*. *UISwipeGestureRecognizerDirection* is the data structure that is used to contain the four different directions recognizable by the class: *UISwipeGestureRecognizerDirectionRight*, *UISwipeGestureRecognizerDirectionLeft*, *UISwipeGestureRecognizerDirectionUp*, and *UISwipeGestureRecognizerDirectionDown*.

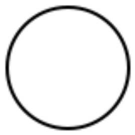

Figure 14. Gesture detected by *UITapGestureRecognizer*

The *UITapGestureRecognizer* class provides the ability to detect when one or more fingers touch a location in the specified view in the interface a specific number of times. The properties that can be set within the class include the *numberOfTapsRequired* and *numberOfTouchesRequired* properties. Both of these properties have a default value of *1*. In order to have an application recognize a gesture that does not conform to any of the properties contained in these Apple-provided classes, a custom gesture recognizer class, which extends the *UIGestureRecognizer* class, must be created. When creating this subclass, the developer must override the *touchesBegan, touchesMoved, touchesCancelled,* and *touchesEnded* functions to achieve the desired results from the gesture recognizer. After these functions have been overridden and the gesture recognizer has been declared, it is then added to the view upon which the gesture should be recognized.

Chapter 5. The GestDefLS Language

The language has been created to allow for the clear and concise definition of custom, multi-touch gestures. It is built to function on top of the state machine that currently exists within Apple's gesture definition framework. Similar to the framework it is built on top of, the language itself also functions as a state machine, with gestures being declared as states that transition from one gesture to the next as the touch requirements for a specific gesture or set of gestures are met. The class diagram for the language can be seen in the figure below.

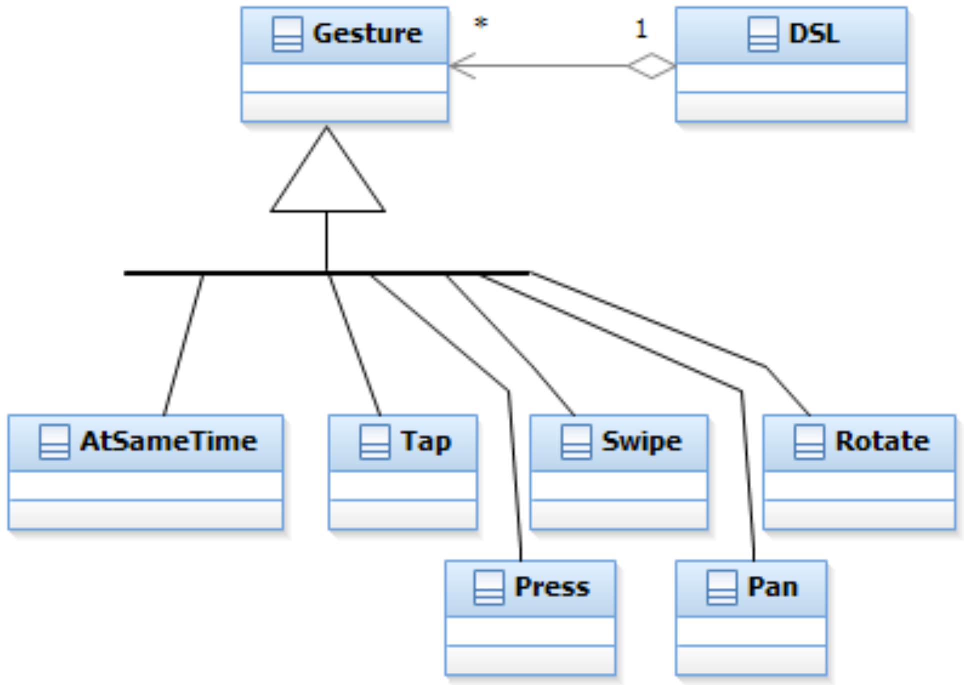

Figure 15. Class diagram for GestDefLS

Extensibility is a positive effect of this design. This would be performed most probably by the user overriding the call to perform gesture detection within the class of the gesture they wish to change. Unfortunately, due to the code within the *DSL* class referencing the simple gestures by type in its computation, the *Gesture* class itself cannot be extended. The code samples for the gesture examples in this chapter can be found in the appendix. Furthermore, the entire repository for the project can be found at https://github.com/ecu-sle-lab/writtenberry-thesis-project.

Gesture Class

A *Gesture* object, the parent class to all other gestures, maintains various pieces of information relating to every gesture, including the function to be executed, the most recent point that has been touched within the current gesture, and a Boolean value to indicate the completion status of the gesture.

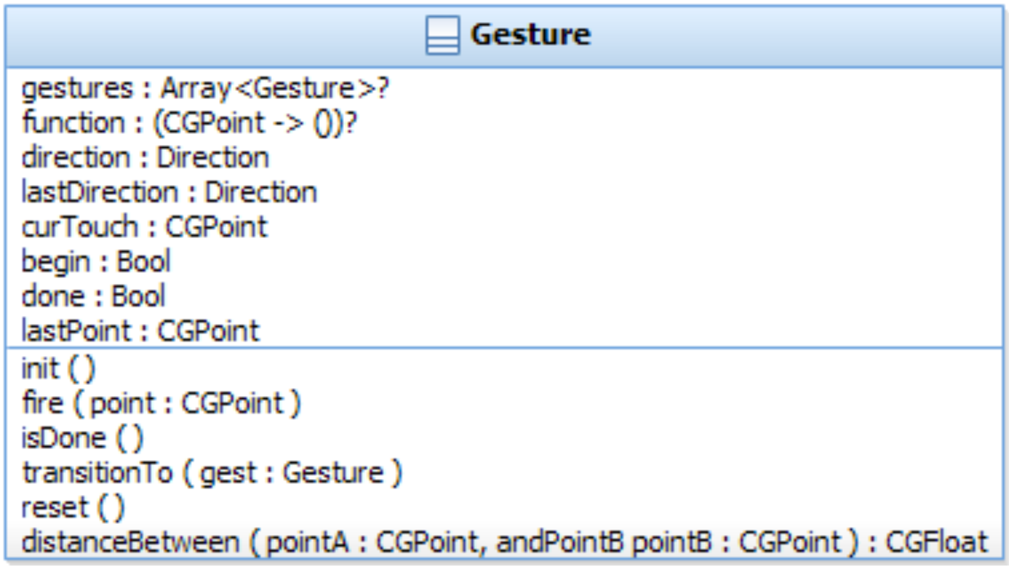
Gesture
gestures : Array<Gesture>?
function : (CGPoint -> ())?
direction : Direction
lastDirection : Direction
curTouch : CGPoint
begin : Bool
done : Bool
lastPoint : CGPoint
init ( )
fire (point : CGPoint)
isDone ( )
transitionTo (gest : Gesture)
reset ( )
distanceBetween (pointA : CGPoint, andPointB pointB : CGPoint) : CGFloat

Figure 16. *Gesture* class diagram

Because Swift provides closures as one of its features, it is possible to store the function processing the gesture as an instance variable. One of the instance variables of the *Gesture* class is an array of *Gesture* objects. This array is set to be of “optional” type. It is only initialized when declaring an *AtSameTime* object, to act as a container for other *Gesture* objects. Also of “optional” type is the *function* instance variable, which stores the function to be called to recognize a specific gesture. Once again, this variable is declared as being of optional type because it is only a characteristic of a traditional *Gesture* subclass, remaining set to nil within an *AtSameTime* object. The Boolean instance variable called *done* is utilized to maintain the completion status if the gesture. This value is initially set to false, and consequently set to true at some point during the call to the gesture's function whenever the condition, or conditions, for a successful gesture are met, or recognized. The *fire* method is implemented to allow a subclass of *Gesture* to run the function stored in its *function* variable. It first checks to see if the gesture has been completed, by looking at the value of the *done* variable. If it has been completed and is still being called (in the case that it is part of an *AtSameTime* object gesture array), the value holding the most recent point touched on the interface is set to the value of the parameter passed to the *fire* method and the function returns. However, if *done* is still equal to false, *fire* calls the value stored in *function*. The next method implemented within the *Gesture* class is the *transitionTo* method. It provides the capability to transition from one gesture to another, copying important information from the instance variables of one gesture into the instance variables of the next gesture to be recognized. If transitioning between two *AtSameTime* gestures, the method iterates through the gesture array in the calling object and copies the important values from each of those individual *Gesture* objects into the individual *Gesture* objects within the gesture array of the *AtSameTime* object passed as a parameter to the *transitionTo* method. If transitioning from one simple gesture to an *AtSameTime* object, only the value of the direction of the single gesture is passed to the *AtSameTime* object, and consequently, to all of the *Gesture* objects within the array of gestures inside the *AtSameTime* object. Finally, a *Gesture* object has a *reset* method that is called to set the values of all the object's instance variables back to what their initial values were when the object was created.

Individual Gesture Subclasses

A handful of actual gestures to be recognized by a single touch are defined as subclasses to the *Gesture* class. These subclasses store values important to the individual gesture, in addition to the function that returns the closure necessary to perform computation of the touches as they move throughout the user interface. These unique instance variables are passed as parameters to the constructor when the user initializes the class within the call that creates the complete complex gesture recognizer. *Pan*, *Press*, *Tap*, *Swipe*, and *Rotate* are the subclasses that comprise the possible gestures.

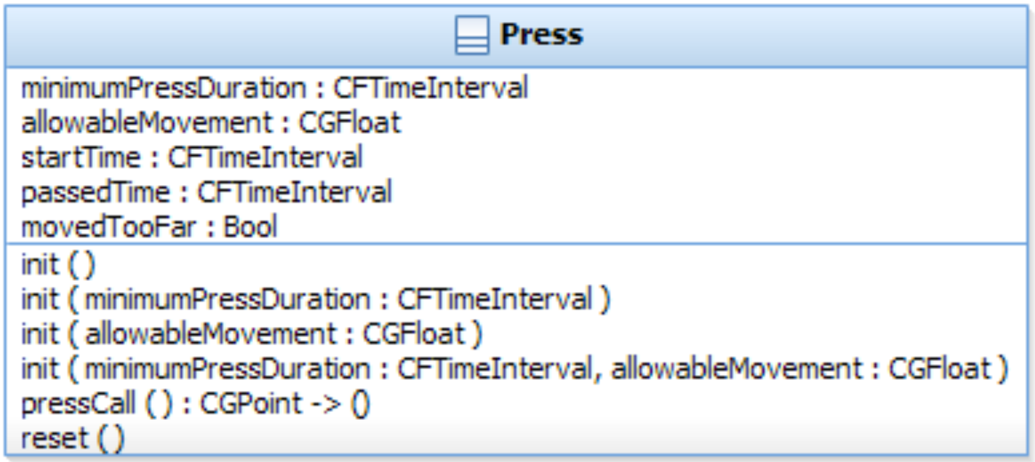
Press
minimumPressDuration : CFTimeInterval
allowableMovement : CGFloat
startTime : CFTimeInterval
passedTime : CFTimeInterval
movedTooFar : Bool
init ()
init (minimumPressDuration : CFTimeInterval)
init (allowableMovement : CGFloat)
init (minimumPressDuration : CFTimeInterval, allowableMovement : CGFloat)
pressCall () : CGPoint -> ()
reset ()

Figure 17. *Press* class diagram

The gesture represented by the *Press* class is computed based off of the amount a time a touch exists in a certain location on the interface. It is only successful if the touch has not moved past the value of *allowableMovement* for at least the amount of time set by *minimumPressDuration*.

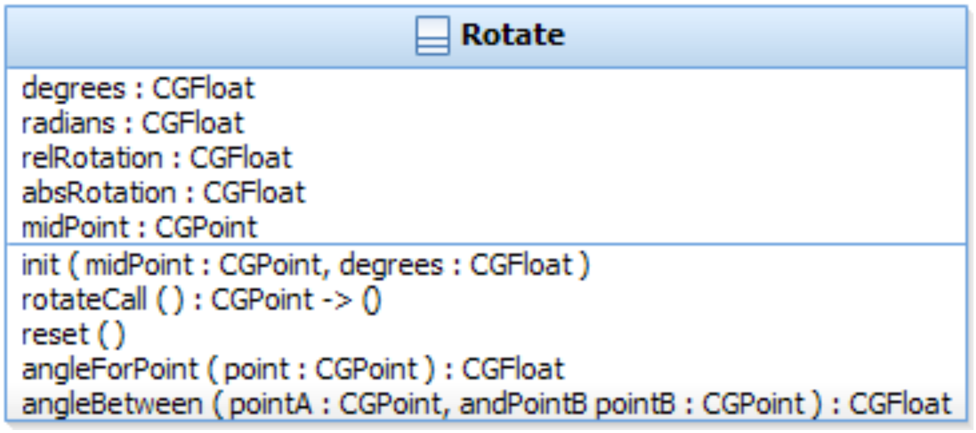
Rotate
degrees : CGFloat
radians : CGFloat
relRotation : CGFloat
absRotation : CGFloat
midPoint : CGPoint
init (midPoint : CGPoint, degrees : CGFloat)
rotateCall ( ) : CGPoint -> ()
reset ( )
angleForPoint (point : CGPoint) : CGFloat
angleBetween (pointA : CGPoint, andPointB pointB : CGPoint) : CGFloat

Figure 18. *Rotate* class diagram

The *Rotate* class performs computation to recognize a gesture where the touch moves a certain angle, indicated by *degrees*, away from the starting point of the touch.

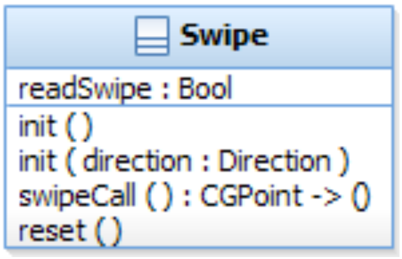

Figure 19. *Swipe* class diagram

A swipe gesture is recognized by computation within the *Swipe* class that determines whether a touch has moved in the direction, indicated by the *direction* variable, away from the starting touch.

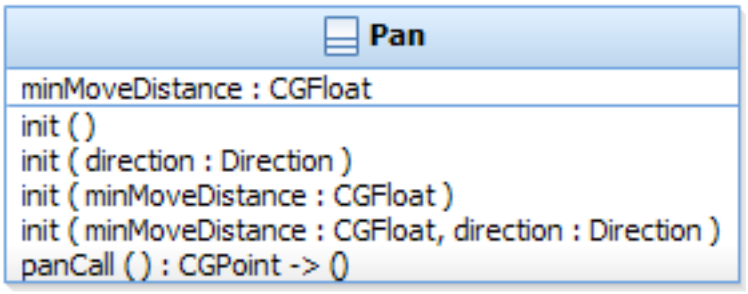
Pan
minMoveDistance : CGFloat
init ( )
init (direction : Direction)
init (minMoveDistance : CGFloat)
init (minMoveDistance : CGFloat, direction : Direction)
panCall ( ) : CGPoint -> ()

Figure 20. *Pan* class diagram

The *Pan* class contains the computation necessary to detect a touch moving a certain distance, *minMoveDistance*, or direction, *direction*, away from the starting touch.

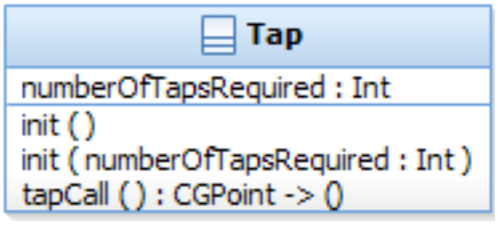
Tap
numberOfTapsRequired : Int
init ( )
init (numberOfTapsRequired : Int)
tapCall ( ) : CGPoint -> ()

Figure 21. *Tap* class diagram

A tap gesture is detected by the *Tap* class, which simply stores the number of taps needed to successfully recognize the tap, in *numberOfTapsRequired*. Unlike the other *Gesture* subclasses, the function stored in this class, *tapCall*, only sets *done* equal to true, because the comparison between number of taps recognized and number of taps required is made in the *DSL* class, without any further computation required.

AtSameTime Gesture Subclass

To facilitate the identification and detection of multiple touches on the interface at one time, the *AtSameTime* class is provided.

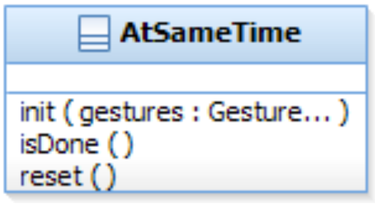

Figure 22. *AtSameTime* class diagram

One of the biggest challenges was deciding how to represent a gesture where one gesture could be performed at the same time as another. Utilizing a "while" type of keyword seemed practical due to its commonness among most programming languages. However, due to its reservation as a keyword within Swift, this was not possible, so it was decided that an *AtSameTime* name would be sufficient, because of its indication that two to five touches should be recognized during the same state as each other before moving on to the next state of the gesture. This class is a unique subclass of the *Gesture* that provides the initialization, completion status, and resetting of a multi-touch gesture. The initialization is performed by a constructor that calls the *init* method of the *Gesture* class, and then stores the *Gesture* array, provided as a parameter by the user, into an instance variable. Completion status is detected by the *isDone* method. Within this method, a temporary Boolean variable is created and set to true, followed by iteration through the array of gestures set by the constructor. If the Boolean *done* variable of any of the gestures within the array are not equal to true yet, then the temporary variable is set to false. Finally, the *AtSameTime* object's done variable is set to the value of this temporary variable. Resetting of the *AtSameTime* object is very similar to the *isDone* method. First the *done* value of the object is set to false, followed by iteration through the object's gesture array, calling the *reset* method on each of those gestures.

DSL Class

The *DSL* class is the catalyst for making the language run and detect the declared complex gesture.

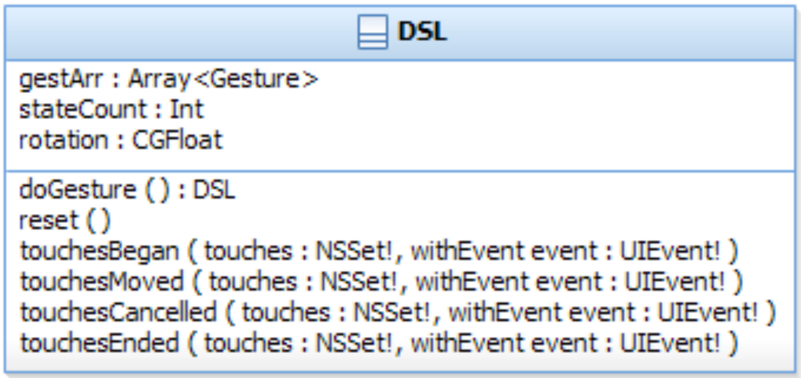
DSL
gestArr : Array<Gesture>
stateCount : Int
rotation : CGFloat
doGesture ( ) : DSL
reset ( )
touchesBegan (touches : NSSet!, withEvent event : UIEvent!)
touchesMoved (touches : NSSet!, withEvent event : UIEvent!)
touchesCancelled (touches : NSSet!, withEvent event : UIEvent!)
touchesEnded (touches : NSSet!, withEvent event : UIEvent!)

Figure 23. ***DSL*** **class diagram**

Its two important instance variables are *gestArr* and *stateCount*. The *gestArr* variable holds an array of *Gesture* objects, the ones that are passed as arguments to the class's *doGesture* method, which are also the gestures that comprise the complex gesture declared by the user. Meanwhile, the *stateCount* variable is used to keep track of the index of *gestArr* that holds the current *Gesture* object on whose function should currently be being called for gesture detection. Naming the variable *stateCount*, rather than *gestCount*, came from the likening of GestDefLS to a state machine. As the main method to initiate the creation of the complex gesture, the *doGesture* method receives all of the user's parameterized *Gesture* objects as input and stores them into the *gestArr* array variable. In addition to this method, the overridden *touchesBegan, touchesMoved, touchesCancelled,* and *touchesEnded* exist within the *DSL* class. In *touchesBegan*, if there are multiple touches on the interface at one time, the location of each touch is assigned to the *Gesture* subclass objects within the array belonging to the first *AtSameTime* object within the array of gestures in the *DSL* class. Next, the assignment of these touch locations are sorted and reassigned based on the location within the interface. The left-most touch in the interface is assigned to the lowest indexed *Gesture*, with that sorting pattern continuing from left to right and lowest to highest array index. Within the *touchesMoved* function, the *fire* method within the *Gesture* object at the *stateCount* index of the array of *Gesture* objects is called, being handled differently depending on whether the object is of type *AtSameTime* or not. If the object is an *AtSameTimeObject*, each of the gestures within that object's array have the *fire* method called until all of the *Gesture* objects in that array have their *done* variable set to true. Otherwise, the object simply has its *fire* method called until its *done* value is set to true. After the currently indexed gesture in the *DSL object's* gesture array has it's *done* value set to true, the variable holding the array index is incremented and the previous *Gesture* object has the *transitionTo* method called on it, with the next gesture in the array being passed as the parameter. Once the array index equals the number of *Gesture* objects within the array, then the state of the *UIGestureRecognizer* is set to *Ended*. Whenever, the gesture ends or is not completely fulfilled on the interface, the *touchesEnded* or *touchesCancelled* functions are called respectively. In the *DSL* class, these functions simply make a call to the *DSL* class's *reset* function, which calls the *reset* method on each of the *Gesture* objects stored in the *DSL* class's array of gestures and sets the *DSL*'s state to *Failed*.

Using GestDefLS

The purpose of GestDefLS is to eliminate the need for difficult-to-understand, hard-coded complex gestures. This succinct gesture definition is accomplished first by the user declaring the gesture recognizer in the traditional way, and then calling a *doGesture* method on the recognizer.

Typical flow of information through the GestDefLS language

1. Gesture recognizer is declared as *DSL* type.
2. The *doGesture()* method is called on the recognizer.
3. Objects of type of subclass of *Gesture* are passed as parameters to *doGesture()* method.
4. If any parameter of *doGesture()* is of type *AtSameTime*, up to five parameters of type of subclass of *Gesture* are passed to that *AtSameTime* object.
5. An array of objects from the parameters of step 3 is created by *doGesture* within the *DSL* class.
6. The computational function within the *Gesture* subclass object at index *stateCount* of the array is called.
7. The *stateCount* variable is incremented on successful detection of a gesture.
8. If *stateCount* equals the number of items in the array, the state of the recognizer is set to *.Ended*.

Figure 24. Flow of information through GestDefLS

Shown in the figure above is the typical execution pattern of a call to the GestDefLS language. In the next section, examples are provided to show how this pattern works with actual gestures.

Example Calls to GestDefLS

An example call to the language can be seen in Example 1 in the appendix. This code creates a recognizer, in the GestDefLS language, for the tickle gesture that is available in *Monkey Pinch* (Begbie, 2014). First, the beginning of the gesture recognizer is declared in the traditional way, declaring values for the *target* and *action* parameters. Next, within the call to the *doGesture* function, three parameters are passed: a *Pan* object with a declared direction of *.Any*, a *Pan* object with a declared direction of *.Opposite*, and another *Pan* object with a declared direction of *.Opposite*. For this particular gesture, the placement and order of touches on the user interface are diagrammed in the figure below. The circle with a single letter inside of it identifies a unique finger that is touching the interface.

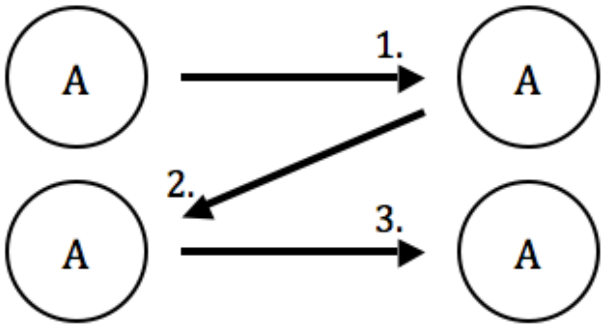

Figure 25. Tickle gesture

When contrasting a GestDefLS call with the code required to create this recognizer without the use of the language, it is clear to see a difference in the amount of required effort. Also shown in Example 1, is the code from the *Monkey Pinch* application, within the *touchesMoved* function of a *UIGestureRecognizer* subclass that would implement the same functionality as the example call to GestDefLS. This code sample omits the overridden *touchesBegan*, *touchesCancelled*, *touchesEnded*, and *reset* functions, which add even more lines of required code to the custom class. Without comments within the source code of the custom class, it can be difficult to tell what exactly the processing code is doing. However, the GestDefLS call can be read as a gesture recognizer being created that requires the performing of a gesture where a single touch must pan in any direction across the view, then pan in the opposite direction from the most recent pan, followed by a pan, yet again, in the opposite direction as the previous pan. The original *UIGestureRecognizer* subclass file required to create the recognizer for the tickle gesture, *TickleGestureRecognizer.swift*, included a total of 81 lines. With GestDefLS, all these lines of code have been reduced to 1 call to the language over the length of 6 lines of code. Another example of the concise nature of GestDefLS is observed when comparing the GestDefLS gesture definition for a "pinch-pan" gesture with the code required to manually implement the gesture. In this example, Example 2 within the appendix, the unique *AtSameTime* class is utilized within the GestDefLS call. Within the first parameter of the *doGesture* method call, the *AtSameTime* object is declared with a *Pan* object with direction of *.Right* and a *Pan* object with direction of *.Left*. Because the location of the parameters to an *AtSameTime* object correspond to their location on the interface, the declaration of the object is saying that the first *Pan* corresponds to the leftmost touch, with the second *Pan* object corresponding to the next touch to the right of the first one. In this case, there are only two parameters to the *AtSameTime* object. Therefore, the motion of a touch on the left moving right and touch on the right moving left creates an inward pinching motion. Similar to the first parameter of the *doGesture* object, the second parameter is an *AtSameTime* object. This one takes two *Pan* objects, each with declared direction of *.Down*, stating that both should detect a downward dragging motion. The diagram of the gesture can be seen in the figure below:

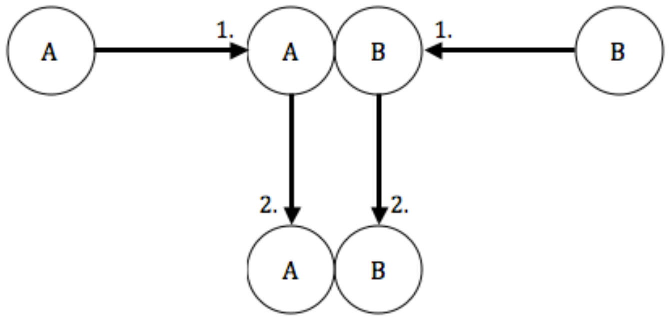

Figure 26. Pinch-pan gesture

The GestDefLS declaration is easier to understand than the custom code because it is, in simple terms, declaring the gesture as doing two pan motions at the same time, followed immediately by two more pan motions at the same time. The custom code, shown beneath the GestDefLS call, comprises only 10 lines of a file that is approximately 100 lines total in length. This can be compared to the one call of 11 lines that it takes to create the gesture recognizer with GestDefLS, which can be even further condensed to less lines of code while still maintaining readability. Yet another example of how GestDefLS saves the developer from writing a lot of code is observed in the one-finger rotate gesture, which is featured in *Circular Knob Demo* (Erholm, 2014). Although, the *UIGestureRecognizer* API provides support for a two-touch rotation gesture, it lacks support for any others. Therefore, this is one thing that GestDefLS provides that is absent from Apple’s toolset. Because of the trigonometry involved in creating such a gesture, even more code is required to create a custom version of it.

The code sample in Example 3 of the appendix is the GestDefLS call required to recognize the one-finger rotation gesture. Within the *doGesture* call of the recognizer, the *Rotate* object is declared. This object requires a *midPoint* parameter, which identifies the point that should be the center that the rotation is being calculated around. The object also requires a *degrees* parameter, which identifies the amount that should be rotated around the mid point for the gesture to be recognized. The figure below identifies, in graphical form, how the one-finger rotation gesture of 90 degrees is performed on the interface.

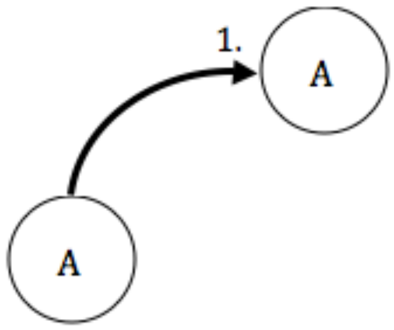

Figure 27. One-finger rotation gesture

There is no code sample for the custom class of this gesture, due to the use of mathematical computation that is outside the scope of this paper. A final gesture to demonstrate GestDefLS is a press-and-drag gesture. The GestDefLS call to create this gesture still requires only one line of code, compared to the 89 lines required to create the gesture in a hard-coded manner. The code sample can be found in Example 4 within the appendix. The motion of the gesture is illustrated in the figure below. It includes one finger performing a press of a designated amount of time, while a second finger is performing a swipe motion at the same time. These are expressed through the use of one *AtSameTime* object in the call using GestDefLS.

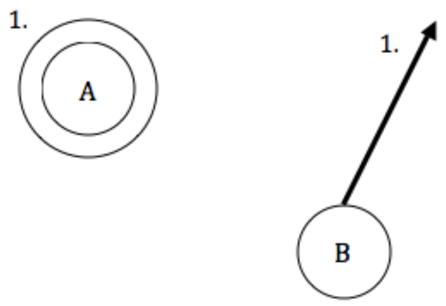

Figure 28. Press-and-drag gesture

The code required to manually create this gesture can be observed in the appendix, as well**.**

State Machine

Gestures executing functions, as transitions from one gesture to the next, make up the basic framework for the state machine of GestDefLS. Because of the presence of the delegate concept within the *UIGestureRecognizer* API, there is no need for one state machine to maintain multiple possible transitions from a single state. Through the use of the *requireGestureRecognizerToFail* method that can be called on a gesture recognizer, two gestures that begin in the same way are handled.

```
recognizer1.requireGestureRecognizerToFail(recognizer2)
```

Figure 29. Example of *requireGestureRecognizerToFail* method

In the figure above, the method call indicates that gesture recognizer *recognizer1* can only continue to be recognized after *recognizer2* has failed. It is the responsibility of the developer to handle any instances where another gesture could possibly begin in the same way as a gesture declared utilizing GestDefLS. Because similar gestures are handled outside of the GestDefLS language, the state machine takes on a very linear structure.

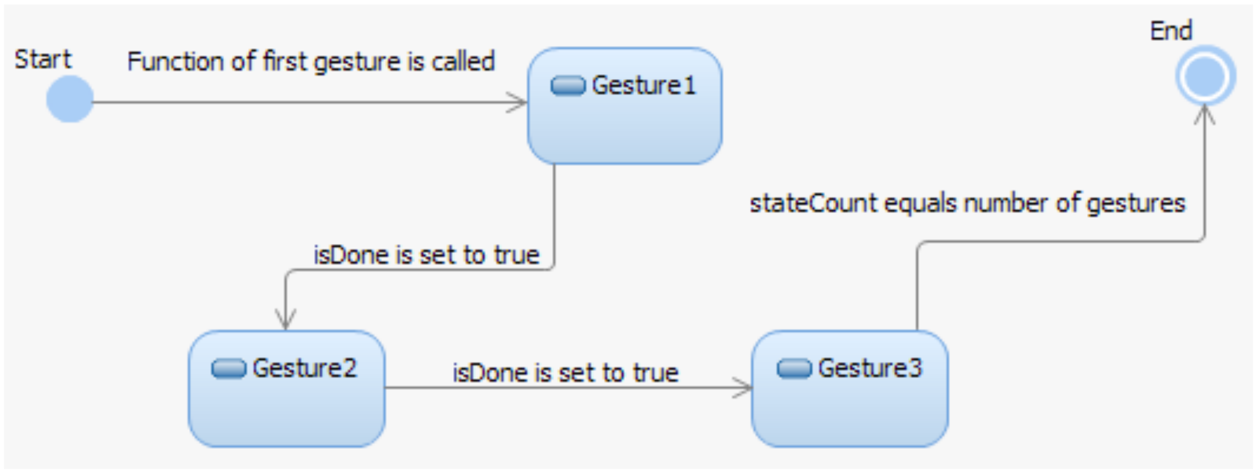

Figure 30. State machine for a one-finger complex gesture

The figure above is the state machine for the successful completion of some generic complex gesture, to be recognized by the DSL, which requires one finger to perform a sequence of three gestures: Gesture1, Gesture2, and Gesture3. The function held by the instance variable in Gesture1 is called to process information about the touch to check if the gesture has been completed yet. This checking continues over and over again, until the *done* variable within the *Gesture* object is set equal to true, to signify a successful detection of the first gesture. The machine transitions to Gesture2. Similar to Gesture1, the function within Gesture2 to process information about the touch is called repetitively, transitioning to Gesture3 whenever *done* is set equal to true. This same process is repeated for Gesture3. However, a counter, called *stateCount*, which increments every time a transition to another gesture occurs, is checked for transition readiness from the last gesture state to the end state. If this counter is equivalent to the number of gestures to be recognized within the complex gesture, then the complex gesture is successful. At this point, the transition from the third gesture state to the end state occurs, setting the state of the gesture recognizer to be *Ended*.

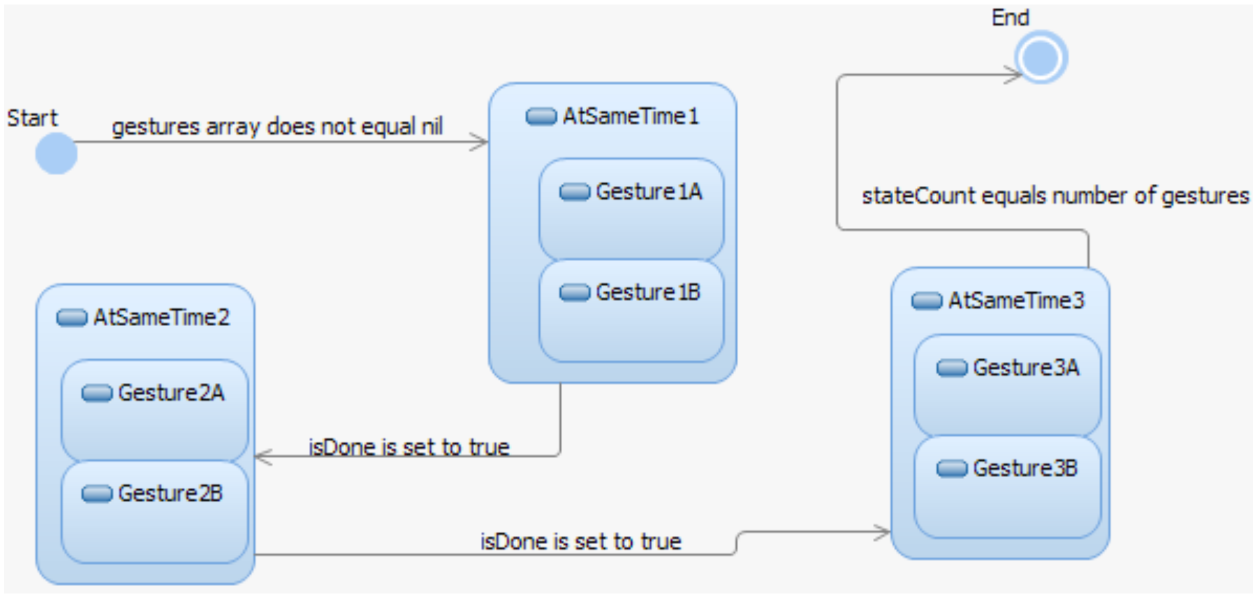

Figure 31. State machine for a multi-finger complex gesture

The above state machine is an example of a complex gesture that requires the successful, incremental completion of gestures that require simultaneous touches by multiple fingers on the interface. If the *gestures* array within the first *Gesture* object is determined to not be equal to *nil*, then the object is determined to be an *AtSameTime* gesture. From this point, the process is similar to the process within the state machine of single-touch gestures. Once the *done* value in the first *AtSameTime* object is set to true, there is a transition to the next *AtSameTime* object. This continues until the final *AtSameTime* object. This transition is also the same for both state machines. At this point, if *stateCount* equals the number of *Gesture* objects within the *DSL* object's *gestArr*, then the state machine transitions to the ending state, where the state of the gesture recognizer is set to *Ended*. In addition to the state machines shown above, the language also provides support for chaining multiple kinds of gestures together, in terms of type and number of gestures. With *one* being identified as a simple gesture, such as a *Pan* or *Tap* object, and *many* being defined as an *AtSameTime* Object, the language provides support for one-to-one, one-to-many, and many-to-many transitions. Many-to-one gesture transitions should also be possible, but have not yet been implemented. Other than this limitation, transitions can also exist from discrete to continuous gestures and vice versa. These possibilities make the language very dynamic in terms of the kinds and number of complex gestures that can be recognized.

Chapter 6. Evaluation

Compared to the other options available for gesture definition languages, GestDefLS is a more ready-to-use option within code. It is also, to our knowledge, the first gesture definition language available in the Swift programming language.

Comparison with DSLs for Gesture Definition

In the figure below, a summary is provided of the comparison of the previously described gesture-based DSLs to GestDefLS.

DSL Name	Similar to GestDefLS	Different from GestDefLS
Midas	• Emphasis on modularization & composition • Attributes (equivalent to properties)	• Gestures defined in a custom lanuage, not the development language • Provides support for spatial gestures
GeForMT	• Constraints (equivalent to properties) • Only provides support for multi-touch gestures	• Gestures defined in a custom lanuage, not the development language • Provides context to simultaneous touches in relationship to each other
GestIT	• *Parallel* operator • Predicates (equivalent to properties)	• Gestures defined in a custom lanuage, not the development language • Provides support for spatial gestures

Figure 32. Similarities and Differences of DSLs to GestDefLS

Midas and GestIT both provide support for *spatial gestures*, those that detect the movement of body parts in space, which is currently outside the scope of the GestDefLS language. However, all three of the languages provide an equivalent to the properties that can be set using named parameters in GestDefLS. The main difference that stands out is the fact that the three DSLs listed in the figure above use a custom language for gesture definition, not the language being used for developing the rest of the application. This can allow more precise definitions of gestures, but it also requires developers to learn an additional language and to use an additional set of development tools. Furthermore, none of these languages currently support translation into Swift. This leaves the developer with three options: implement a translation from one of these languages into Swift themselves; write custom gesture-recognition code using the *UIGestureRecognizer* framework; or use the GestDefLS language to define complex gestures directly in Swift. The obvious benefit of using GestDefLS over one of the available DSLs is the time and effort saved from not having to perform additional implementation.

Comparison with Swift

One of the main benefits of using GestDefLS over the *UIGestureRecognizer* API is the difference in the number of lines of code required to define a gesture in traditional Swift code extending the *UIGestureRecognizer* class versus the number of lines required to define a gesture using GestDefLS. This difference can be observed in the figure below:

Gesture	**Swift code**	**GestDefLS code**
Tickle	81	6
Pinch-pan	97	11
One-finger rotation	178	4
Press-and-drag	89	7

Figure 33. Comparison of line counts in Swift vs. GestDefLS

There is a substantial difference, with an average of 104 lines of code being eliminated, per gesture, using GestDefLS over traditional Swift code. Furthermore, because the GestDefLS calls in this paper have been declared over multiple lines of code for readability purposes, the lines of code for these calls can be trimmed down even more depending on how the developer wishes to declare them. This elimination of lines of code, combined with improved modularity, results in code that is easier to understand. Another benefit of this is that identification of problem areas in code can be isolated to a smaller section of the application. Consistency in regards to the types of properties available within the GestDefLS *Gesture* subclasses compared to those available with the *UIGestureRecognizer* subclasses was also a goal of the GestDefLS language.

Apple *UIGestureRecognizer* subclasses	**GestDefLS *Gesture* subclasses**
UITapGestureRecognizer Properties: *numberOfTapsRequired* *numberOfTouchesRequired*	***Tap*** Properties: *numberOfTapsRequired*
UISwipeGestureRecognizer Properties: *direction* *numberOfTouchesRequired*	***Swipe*** Properties: *direction*
UIRotationGestureRecognizer Properties: *rotation* *velocity*	***Rotate*** Properties: *midpoint* *degrees*
UILongPressGestureRecognizer Properties: *minimumPressDuration* *allowableMovement* *numberOfTouchesRequired* *numberOfTapsRequired*	***Press*** Properties: *minimumPressDuration,* *allowableMovement*

UIPanGestureRecognizer <u>Properties:</u> *maximumNumberOfTouches* *minimumNumberOfTouches*	***Pan*** <u>Properties:</u> *minMoveDistance* *direction*
UIPinchGestureRecognizer <u>Properties:</u> *scale* *velocity*	**No standalone equivalent** **(must be composed of two *Pan* objects within an *AtSameTime* object)**

Figure 34. Comparison of available properties within gesture classes

As seen in the figure above, most properties of the *UIGestureRecognizer* subclasses map directly to those properties available within the GestDefLS *Gesture* subclasses, in regards to their name and purpose in defining the gesture. This provides the benefit of the developer having a familiar experience, in terms of kinds of available gestures, when declaring gestures with GestDefLS. For *Tap, Swipe, Press,* and *Pan*, the properties relating to number of touches have not been included. This property is implied by the definition of a GestDefLS gesture, with the number of required touches being inferred based upon how many times a certain type of *Gesture* object is passed to the same *AtSameTime* object, if not being declared as a single touch. Meanwhile, the *velocity* property within the *UIRotationGestureRecognizer* class has been deemed future work to be added to the *Rotate* class during the next iteration of the project. The *Pan* object has been given additional properties to calculate distance and direction, in an effort to increase the variety of gestures available.

Limitations

Although a sufficient beginning to a multi-touch declaration language, GestDefLS is not without its limitations. One such limitation is the low effectiveness in terms of modularity when needing to transform an image, using some kind of gesture, like a rotation or pan gesture, for example. In the case of a rotation gesture, after declaring the gesture with GestDefLs, the developer still has to manage the transformation and know how to set the midpoint to the center of the view that the gesture recognizer is attached to. Another limitation is that, within an *AtSameTime* object, discrete gestures cannot be mixed with continuous gestures as parameters. This is due to the nature of the gesture recognizer. Most of the handling for discrete gestures is performed in *touchesBegan* and *touchesEnded*, with continuous gestures being handled mostly by the *touchesMoved* function. A possible solution to combining the two types of gestures requires further exploration. When discovered, this solution could increase the number of combinations of complex gestures available to the developer. Another feature lacking within the language is the ability to declare a gesture that transitions from an *AtSameTime* gesture to a single, simple gesture. The difficulty with the problem is determining how to handle the passing of location and direction information from multiple, simultaneous gestures to a single one, in a way that would be practical and make sense in terms of transitioning from one gesture to the next. Although perceived to be feasible, it has not yet been implemented successfully. A *merge* feature has been proposed as a possible solution to this problem. The implementation of this feature would also increase the number of gestures available to be detected by the language.

Chapter 7. Conclusion and Future Work

The gesture definition tool described in this paper is a means of expressing gestures in a concise and easy to understand manner. Specifically, the tool aids in the process of defining complex or unique gestures that are otherwise complicated to express with regular Swift code and Apple APIs. It provides a fairly good range of customization. Moving forward, it is desired that GestDefLS function through the use of a fluent interface, which is a goal for future iterations of the language. This type of design would improve the readability of a call to GestDefLS. Also, It would be ideal to add more customization options, in terms of parameters passed in to an individual gesture object, thus giving the developer even more control over the kinds of gestures they are defining. Another goal in the future is providing implementations of the tool on other platforms, such as the Android operating system, and even providing a framework for GestDefLS that can be easily transformed to other languages.

References

Apple Inc. (2013, January 28). Event Handling Guide for iOS. Cupertino, CA, USA: Apple, Inc.

Apple Inc. (2014). *The Swift Programming Language.* Cupertino, CA, USA: Apple Inc.

Begbie, C. (2014). *Monkey Pinch [Software].* Retrieved from http://www.raywenderlich.com/76020/using-uigesturerecognizer-with-swift-tutorial

Erholm, S. (2014). *Circular Knob Demo [Software].* Retrieved from https://github.com/CayuseConcepts/KnobRotation

Fowler, M., & Parsons, R. (2011). *Domain-Specific Languages.* Boston, MA, USA: Addison-Wesley.

Hoste, L., & Signer, B. (2014). Criteria, Challenges and Opportunities for Gesture Programming Languages. *1st International Workshop on Engineering Gestures for Multimodal Interfaces* (pp. 22-29). Rome, Italy: Lode Hoste and Beat Signer.

Illusion Labs AB. (2008). *Touchgrind [Software].* Retrieved from http://www.touchgrind.com/skate/

Kammer, D., Henkens, D., Henzen, C., & Groh, R. (2013). Gesture Formalization for Multitouch. *SOFTWARE – PRACTICE AND EXPERIENCE* , 527-548.

Kammer, D., Wojdziak, J., Keck, M., Groh, R., & Taranko, S. (2010). Towards a Formalization of Multi-touch Gestures. *ITS'10* (pp. 49-58). Saarbrücken, Germany: ACM.

Le Goaer, O., & Waltham, S. (2013). Yet Another DSL for Cross-Platforms Mobile Development. *Proceedings of the First Workshop on the globalization of domain specific languages* (pp. 28-33). Montpellier, France: ACM.

Nuñez-Valdez, E. R., Sanjuan, O., Garcia-Bustelo, B. C., Cueva-Lovelle, J. M., & Hernandez, G. I. (2013). Gade4all: Developing Multi-platform Videogames based on Domain Specific Languages and Model Driven Engineering. *International Journal of Artificial Intelligence and Interactive Multimedia , 2* (2), 33-42.

Scholliers, C., Lode, H., Signer, B., & De Meuter, W. (2011). Midas: A Declarative Multi-Touch Interaction Framework. *ACM Conference on Tangible, Embedded and Embodied Interaction* (pp. 49-56). Funchal, Portugal: ACM.

Spano, L. D., Cisternino, A., Paternò, F., & Fenu, G. (2013). GestIT: A Declarative and Compositional Framework for Multiplatform Gesture Definition. *EICS’13* (pp. 187-196). London, England: ACM.

Steiner, D., Turlea, C., Culea, C., & Selinger, S. (2013, January 1). Model-Driven Development of Cloud-Connected Mobile Applications Using DSLs with Xtext . *Lecture notes in computer science* , 409-416.

Appendix: Selected Code Samples

Example 1. Tickle Gesture:

GestDefLS call

```
let recognizer = DSL(target: self, action: Selector("handleGesture:"))
        .doGesture(
                   Pan(direction: .Any),
                   Pan(direction: .Opposite),
                   Pan(direction: .Opposite)
         )
```

Swift code (excerpt from touchesMoved() in *TickleGestureRecognizer.swift*)

```
if abs(moveAmt) < self.distanceForTickleGesture {
        return
}

if self.lastDirection == .DirectionUnknown || (self.lastDirection == .DirectionLeft && curDirection ==
.DirectionRight) || (self.lastDirection == .DirectionRight && curDirection == .DirectionLeft) {
        self.tickleCount++
        self.curTickleStart = ticklePoint
        self.lastDirection = curDirection

        if self.state == .Possible && self.tickleCount > self.requiredTickles {
                self.state = .Ended
        }
}
```

Example 2. Pinch-Pan Gesture:

GestDefLS call

```
let recognizer = DSL(target: self, action: Selector("rotation:"))
        .doGesture(
         AtSameTime(gestures:
                 Pan(direction: .Right),
                 Pan(direction: .Left)
         ),
         AtSameTime(gestures:
                 Pan(direction: .Down),
                 Pan(direction: .Down)
                 )
         )
```

Swift code (excerpt from touchesMoved() in *PinchDrag.swift*)

```
if (abs(moveAmt) > self.distanceForTickleGesture && ((curDirection == .DirectionLeft && curDirection2 ==
.DirectionRight) || (curDirection == .DirectionRight && curDirection2 == .DirectionLeft))) {
        var distanceForDragDownGesture:CGFloat = 25.0
        let moveYAmt = newTouch.y - newTouch2.y
        let moveYAmt2 = curTouch.y - curTouch2.y

        if abs(moveYAmt) < distanceForDragDownGesture {
                return
        }
        self.state = .Ended
}
```

Example 3. One-finger rotation Gesture:

GestDefLS call

```
let recognizer = DSL(target: self, action: Selector("rotation:"))
        .doGesture(
            Rotate(midPoint: view.convertPoint(view.center, fromCoordinateSpace: self.view), degrees: 90)
        )
```

Example 4. Press-and-drag Gesture:

GestDefLS call

```
let recognizer = DSL(target: self, action: Selector("handleGesture:"))
        .doGesture(
                AtSameTime(gestures:
                        Press(),
                        Pan()
                )
        )
```

Swift code (excerpt from touchesMoved() in *MoveWhilePressed.swift*)

```
if touch!.count == 2 {
          if self.press == touch![0].locationInView(self.view) {
                    self.press = touch![0].locationInView(self.view)
                    self.newTouch = touch![1].locationInView(self.view)
          } else {
                    self.press = touch![1].locationInView(self.view)
                    self.newTouch = touch![0].locationInView(self.view)
         }
         if prevTouch != CGPointZero {
                    self.distanceTraveled = self.distanceTraveled + distance(prevTouch, and: self.newTouch)
          }
} else {
         self.press = touch![0].locationInView(self.view)
         self.distanceTraveled = 0.0
}

let moveAmt = press.x - prevPress.x
durationOfPress = currentTime - startTimeAbsolute

if abs(moveAmt) < maxPressMovement {
          if(self.distanceTraveled > self.distanceForTickleGesture && durationOfPress >
          minimumPressDuration)
                    self.state = .Ended
          } else {
                    return
          }
}
```