A SOFTWARE DEVELOPMENT KIT FOR CAMERA-BASED GESTURE RECOGNITION

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by Robert Devlin Cronin

December 2013

COMMITTEE MEMBERSHIP

TITLE:	A Software Development Kit for Camera-Based Gesture Recognition
AUTHOR:	Robert Devlin Cronin
DATE SUBMITTED:	December, 2013
COMMITTEE CHAIR:	Franz Kurfess, PhD Professor of Computer Science
COMMITTEE MEMBER:	Aaron Keen, PhD Professor of Computer Science
COMMITTEE MEMBER:	Gene Fisher, PhD Professor of Computer Science

ABSTRACT

A Software Development Kit for Camera-Based Gesture Recognition

Robert Devlin Cronin

Human-Computer Interaction is a rapidly expanding field, in which new implementations of ideas are consistently being released. In recent years, much of the concentration in this field has been on gesture-based control, either touch-based or camera-based. Even though camera-based gesture recognition was previously seen more in science fiction than in reality, this method of interaction is rising in popularity. There are a number of devices readily available to the average consumer that are designed to support this type of input, including the popular Microsoft Kinect and Leap Motion devices.

Despite this rise in availability and popularity, development for these devices is currently an arduous task, unless only the most simple of gestures is required. The goal of this thesis is to develop a Software Development Kit (SDK) with which developers can more easily develop interfaces that utilize gesture-based control. If successful, this SDK could significantly reduce the amount of work (both in effort and in lines of code) necessary for a programmer to implement gesture control in an application. This, in turn, could help reduce the intellectual barrier which many face when attempting to implement a new interface.

The developed SDK has three main goals. The SDK will place an emphasis on simplicity of code for developers using it; will allow for a variety of gestures, including gestures made by single or multiple trackable objects (e.g., hands and fingers), gestures performed in stages, and continuously-updating gestures; and will be device-agnostic, in that it will not be written exclusively for a single device. The thesis presents the results of a system validation study that suggests all of these goals have been met.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1: Introduction

The question of which form of human-computer interaction is superior to others is one which has proven perennial, and is unlikely to be answered in the immediate future. Throughout the lifetime of the computer, we have seen a variety of different interaction methods and devices. Interaction methods span from directly interfacing with the machine's programming via program cards to using a commonplace device setup like a mouse and keyboard to methods like 3-dimensional gesture interaction which were previously only found in the domain of science fiction.

As we continue to develop different methods of interaction, a number of areas for improvement arise. One of the most important and conspicuous areas in need of advancement is in the basic technology and hardware itself - in order to have any form of interaction, the hardware must be able to support the interaction accurately. Another area which will need to be improved is that of the software which interfaces between the hardware and the user. Without this software, there is very little use for even the most advanced hardware.

However, there is also a third area which is sometimes neglected. Good development tools and environments frequently lag behind cutting-edge technology. In order for any interaction method to be fully adopted, it is important to not only have the hardware capabilities and software support for basic interaction, but also the developmental abilities to expand the use of the device. Without this ability, a device is incredibly limited - it is doomed to the purgatory of stagnant development.

With more advanced interaction methods continually emerging, the need for simple and powerful development tools becomes more important. Whereas a mouse has a relatively simple feature set of x-y coordinates with some number of buttons, most modern interaction methods have a much stronger focus on gestural movements, either in 3-dimensional space or upon a touch screen.

Despite this emerging need, we see that, with a few notable exceptions, relatively few tools exist that support this development in the simple-yet-powerful manner which is always desirable. Towards this end, we strive to create a new Software Development Kit (SDK) which supports customizable gesture interaction in 3-dimensional space for use with camera-based gesture-recognition devices.

Section 2 of this thesis explores a short history of the popular human-computer interaction methods, particular those with a form of gesture recognition. Section 3 explores some of the similar work which has been done in the field of SDKs for gesture recognition on camera-based devices. Section 4 describes the primary goals of the SDK which we will develop, while section 5 explains the device chosen for development. Section 6 explores in detail the implementation of the SDK. Section 7 describes the evaluation requirements for the system, and the results from this evaluation. Section 8 illustrates sample use of the system. Section 9 outlines future work for this project. Finally, section 10 concludes this thesis.

Chapter 2: History and Background

This section provides a brief overview of the history and background of the different technologies that have been popular in human-computer interaction. Though camera-based visual gesture interaction has been one of the primary concentrations in recent development, many other devices were created for gesture-based interaction (though the type of gesture has varied).

2.1 Pointing Devices and Keyboard

The mouse-and-keyboard method of human-computer interaction, which has been around for decades, still persists as one of the most common forms of HCI, as can be seen anytime one sits down at a personal computer. The keyboard predates the computer, being a carryover from the typewriters of the 1800s. The computer mouse emerged as an HCI interface at a much later date, with the first mouse design patented in 1970 by Douglas Engelbart. [3] Though these devices have a number of problems and limitations, they have persisted and remain one of the most dominant forms of interaction. [20][14][7]

Though the keyboard has remained largely the same over the years, there are now a number of different devices which operate in similar ways to a computer mouse, serving as a pointing device. In 1989, Wacom Co. was awarded a patent for their "position detecting apparatus"; this closely resembled a tablet with a stylus. [21] Following this, in 1990, a patent was published for the trackball mechanism, which has since been integrated into computer mice. [15] Finally, in 1994, the first patent was awarded for a touchpad. [5] These devices all function in conceptually similar ways.

The use of a pointing device and keyboard is fairly commonly understood in today's technological world. The keyboard is used to enter textual information in a quick and efficient manner (with experienced users able to produce well over one hundred words per minute). The keyboard can also be used to manipulate certain objects, depending on the context (for instance, pressing a left-arrow in a mapping application will likely shift the map view). The pointing device is moved in order to adjust the x-y coordinates of the on-screen pointer, and buttons on the pointing device, if present, have different effects in different contexts.

Combinations of pointing devices and keyboards have many different limitations. The primary limitation of a keyboard is that the device is confined to a very finite number of atomic inputs (specifically, the number of keys on the keyboard, or key combinations, if supported). Pointing devices, while able to produce a greater number of inputs (as a "click" at a different point in the screen may have a different effect, each location can be counted as a separate input option), are still limited to a two-dimensional X-Y plane. This makes interacting with 3-dimensional objects (such as those found in the real world in everyday life) inherently difficult and unintuitive. Additionally, mouse-and-keyboard interaction does not support the same style of natural gesture that many other technologies use.

2.2 Touchscreen

Perhaps the most common form of human-computer interaction after the traditional mouse-and-keyboard style is that of the touchscreen. Touchscreens, like mice and keyboards, have been around for many decades. The idea was originally described by E.A. Johnson in 1965, and again, more fully, in 1967. [8] Touchscreens have continually gained popularity over the years, and are now ubiquitous in new technology. For instance, touch screens may be found very commonly (nearing exclusively) in all new tablets and smart phones, and are continuing to gain ground in personal computing (with more new desktop monitors supporting touch screen capabilities, and operating systems like Windows 8 being optimized and designed with touchscreen interfaces in mind).

Touchscreen devices are used, as their name implies, through direct contact with the screen of the device. Modern versions of these devices support gestural input; an example of this gestural input is that of a common “zoom” gesture by placing two fingers at different points in the screen and spreading the fingers farther apart (intuitively, this is similar to "pulling" the area closer to the user or "stretching" the area out to occupy a greater space). [19]

While touchscreen devices are an important step towards gestural input, these devices have the same two-dimensional limitation as the mouse. Thus, while it may be conceptually understandable to interact with three-dimensional objects with a touchscreen device, it will not be as intuitive as if the gestures themselves could be performed in three-dimensional space. Additionally, with touchscreen devices, the precision required in movements can be prohibitive for some users, due to the small size of the input controls (we see this particularly with compact devices such as cell phones).

2.3 Nintendo Wii and Playstation Move

The Nintendo Wii was originally released in 2006, and with it came the Wii Remote. [19] This device provided the general public with a product which was both commercially-available and affordable, in addition to touting (at the time) advanced capabilities for motion tracking. As such, it was quickly adopted into many different projects seeking such technology, including focuses such as finger- and object-tracking, interactive displays, head tracking, augmented reality, and others. [11]

The Playstation Move was released four years after the Nintendo Wii, in 2010. [19] This controller provided very similar capabilities as Wii remote, allowing for advanced motion tracking and related technologies. However, possibly due to the relatively late release and lack of any substantial additional features, it does not appear that the Playstation Move was as popular as the Wii Remote in research and academia; this assumption is based on the very limited number of scholarly papers relating to projects with the Playstation Move.

Both the Nintendo Wii Remote and Playstation Move are used in a very similar fashion. The general concept is that the user will grip the controller in his or her hand, and will move with the remote. Since the remote has IR sensors and an accelerometer, it is able to determine with some accuracy the motion that is being performed. These technologies also allow the user to use the remote as both a pointing device, as well as a motion tracker.

While the presentation of the Wii Remote and Playstation Move make several important steps - most importantly, presenting relatively novel technology to the public at an affordable price - they are not without their limitations. Though they do remove the two-dimensional limitation of touchscreens and mice, they introduce a new, and very constrictive, limitation: all information is gathered through the remote. This introduces a number of problems, but the most prominent of these can be expressed as the prevalence of a kind of false negatives and false positives.

A false negative, in this sense, may occur when the user performs a gesture which does not heavily incorporate the hand holding the remote - if the remote does not move, the gesture is not registered. This greatly limits the number and variety of gestures. A false positive may similarly occur when only the hand holding the remote moves, but the gesture is defined to require greater movement. As an example, if the desired gesture involves an arm-swinging motion, flicking the wrist has a similar effect. This also makes it difficult to differentiate between different possible gestures.

2.4 Sensor Gloves

Sensor gloves are aptly-named devices which are worn on the user's hands and provide input about the actions performed. The first sensor glove was proposed by Thomas Zimmerman, and patented in 1985. [22][23] Since, they have developed and advanced, and continue to be available to general consumers, even if they are not as prevalent as other human-computer interaction devices. [20]

The use of sensor gloves is as would be expected: they provide information regarding the actions performed by the user's hands. Since the device is a glove, rather than a remote, the user is able to maintain the majority of the dexterity in his or her hands, allowing for far more complex actions to be performed.

Similar to the Wii Remote or Playstation Move, sensor gloves and similar devices are concentrated in a particular area - the hands (though unlike the remotes, this choice is by design). However, this is partially (or even fully) compensated for because of the dexterity which is retained by the user and can be seen by the glove. For instance, with the appropriate device, this technology has used even for something as so precise as recognizing sign language. [9] This is not, though, a full substitute for a user being able to his or her entire body in gesture recognition. Additionally, the use of a glove, while less constrictive than being required to hold an object (like a remote), is still more constrictive than is natural.

2.5 Microsoft Kinect

The Microsoft Kinect was released in 2010, providing a technology to consumers which had previously been unavailable or unrealistic. [19] Unlike other devices, the Kinect can be used with the user's entire body and does not rely on any devices other than the Kinect's cameras and sensors. It has been used extensively in research areas, including medical, graphical, and further HCI research. [2][6][18][4]

The Kinect does not require any components other than the cameras and sensors it contains in the main unit; i.e., the user does not have to wear any additional items for the

Kinect to recognize the user's actions. Additionally, the user has full control of his or her body in the gestures and actions performed. The Kinect will capture the user's movement so long as the movement is large enough and within the Kinect's (generous) viewable field.

The Kinect has fewer limitations than most gesture-based interaction devices. However, it does still have one major drawback: accuracy. While the Kinect is reasonably accurate, it is not so accurate that a user who meant to perform one action could not feasibly accidentally perform another, if the actions are similar enough. In particular, any gesture where the analysis of the user's hands or fingers is important is very difficult to recognize with the Kinect hardware.[1]

2.6 Leap Motion

The Leap Motion is a new device which strives to incorporate the dexterity of the sensor glove technology, without the requirement of gloves. Additionally, the Leap Motion is said to have accuracy one hundred times greater than that of any other gesture-recognition device (such as the Kinect). [13] Since this is a recent technology, it has not been used in a very large number of applications.

The Leap Motion controller is used by gesturing with hands and fingers above the device. The user is not required to wear any gloves or other devices; the interaction is done entirely through the device using depth-based sensors in order to determine the movement of the user.

[1] This applies to the original Microsoft Kinect. The upcoming version, the Kinect One, is said to have significant improvements; however, at the start of this project, it was unreleased.

The Leap Motion improves greatly upon the formula of sensor gloves, while presumably maintaining the accuracy. Unfortunately, the Leap Motion does support a relatively constricted space in which to work; the site currently advertises eight cubic feet of perceived space. Additionally, full use of a user's body is still prohibited. However, these are intended restrictions, which may or may not be entirely negative.

2.7 Others

There are a number of other Human-Computer Interaction devices and methodologies which have not been discussed here. The goal of this section has been to serve as background and to track the evolution towards gesture-based interaction; as such, certain technologies which have not seen wide-spread use are not discussed. Similarly, technologies which do not relate to gesture-based interaction (such as voice commands), are not discussed.

Chapter 3: Similar Work

The focus of this thesis is to create an SDK for camera-based gesture recognition. Of the devices mentioned above, the Microsoft Kinect and the Leap Motion each fit this requirement. Both of these devices have official SDKs, which are developed, maintained, and released by the manufacturer of the device (Microsoft and Leap Motion, respectively). Additionally, the Kinect has a number of third-party SDKs available. Each of the available SDKs are discussed briefly below.

3.1 Existing Microsoft Kinect SDKs

Given the popularity of the Kinect sensor, it it unsurprising that there are a variety of different SDKs available. Among these SDKs are the official Microsoft Kinect for Windows SDK, the OpenKinect project, Code Laboratories NUI platform (CL NUI), and OpenNI (and its accompanying middleware, NiTE).

3.1.1 Microsoft Kinect for Windows SDK

The Microsoft Kinect for Windows SDK is the officially-sponsored SDK for the Kinect device, and is developed and supported by the Microsoft Corporation (which also sells the Kinect device). Though this is the official SDK for the device, it was not the first SDK released. The Microsoft Kinect for Windows SDK was released eight months after the Kinect's own November 2010 release, placing the SDK's release in June, 2011. This SDK is significantly different from the others, given the fact that it is official and developed by the core manufacturer of the device; this provides both advantages and disadvantages. [12]

The advantages of the Microsoft-developed SDK center around the idea that, as it is released by the manufacturer of the device, the SDK can be far more easily optimized. Other, third-party SDK developers may not have access to all the necessary information in order to optimize code for the specific hardware, or may not be cognizant of any impending changes to the hardware. Additionally, as a large corporation, Microsoft has the resources to devote to the development of a high-quality SDK for the Kinect. [12]

The disadvantages of the Kinect for Windows SDK are all focused around the intent of Microsoft to target specific platforms, since the company has a stake in the platform, as well as the device. As such, the SDK only allows development for applications to run on Windows platforms. Additionally, a Visual Studio development environment is required, and the SDK only supports C++, C\#, and Visual Basic languages. [12]

3.1.2 OpenKinect

As mentioned above, at the release of the Kinect device, there was no official SDK available. As such, developers hoping to create applications for the new device had to find an alternative in order to interact with the Kinect's sensors. A contest, sponsored by Adafruit Industries, offered a prize to the first developer of an open-source driver set for the Kinect device. The winner of this contest was OpenKinect. OpenKinect is a free, open-source, and primarily community-developed SDK for the Kinect device. [16]

There are a number of advantages with the OpenKinect SDK. The first is that it is free and open-source. As such, any licensing or platform issues which may have been present in the Microsoft SDK are avoided. Additionally, significantly more languages are supported with the OpenKinect project, including C++, C\#, Java, JavaScript, Python, and others. In this way, the SDK is significantly more versatile than the official Microsoft offering. [16]

The OpenKinect SDK is not without its downfalls. The most significant of these appears to be a general lack of completeness and cohesiveness in the project; this is most likely due to the community-driven nature of the project. This incompleteness manifests itself most apparently in the cases when features and capabilities of the Kinect are not fully implemented (such as voice recognition). This frequently also translates into a less-powerful implementation of those features which are present. [16]

3.1.3 Code Laboratories NUI Platform

Similar to the OpenKinect, the Code Laboratories Natural User Interface SDK came out of the contest to open-source the Kinect's capabilities. Though the CL NUI SDK was allegedly released prior to the OpenKinect SDK, the author did not open-source the code, leading to a loss in the competition; the CL NUI SDK was later officially released on December 8th, 2010. [1]

The main advantage of the CL NUI SDK is that it is open-source, and it supports a greater variety of languages than the Microsoft SDK (including C, C++, C\#, and Java). Unfortunately, it appears that the CL NUI project has been discontinued, and is no longer being supported or updated. Furthermore, the CL NUI SDK does not support as wide a variety of languages as the OpenKinect SDK, and also shares the requirement with the Microsoft SDK for a Windows platform and Visual Studio development. [1]

3.1.4 OpenNI and NiTE

OpenNI is an open source SDK targeting NUI development through 3D sensing devices, including the Kinect. However, unlike all the other SDKs mentioned, this library is not limited to the Kinect. Another key device which is targeted by the OpenNI SDK is the PrimeSense Sensor. The Kinect uses technology developed by the PrimeSense company in the Kinect, and thus, these two devices operate very similarly, but they are not identical. OpenNI also includes support for a middleware library called NiTE - a natural interaction library created by PrimeSense. NiTE versions 1.x were freely available; however, versions 2 and above are not. [17]

OpenNI has a number of advantages. Similar to OpenKinect, OpenNI supports multiple platforms and is freely available (as is NiTE, up to version 2). It also supports a wider variety of languages than the Windows SDK, including C++, C\#, and Java, as well as a wider variety of operating systems. Unlike the Windows SDK, a Visual Studio development environment is not required. Additionally, the OpenNI SDK shares two advantages with the Windows SDK which other SDKs have been lacking - the backing of a large company, and the relation with the product. Given that OpenNI is very heavily tied with PrimeSense, the company which manufactures the device responsible for the depth sensors in the Kinect (the primary sensor in gesture recognition), OpenNI has a great deal of power behind it. [17]

The disadvantages of the OpenNI SDK are that it is not quite as versatile as the OpenKinect SDK as far as language support is concerned. Though it supports most of the popular, "high-level" languages, it has the distinct disadvantage of not supporting any scripting languages such as Python or Javascript, which can be favored for rapid development. [17]

3.2 Existing Leap Motion SDKs

Unlike other motion sensors, currently, there is only one available software developer kit for the Leap Motion. This SDK is the official kit released by the developers, and has a variety of different capabilities. There are two portions of the SDK which are worth addressing: the gestures which are currently recognizable, and the tracking data which can be accessed from the Leap Motion. [10]

3.2.1 Gestures

The Leap Motion SDK includes the functionality to recognize a number of gestures immediately with no extra configuration. [10]

3.2.1a Gesture Vocabulary

At the time of this writing, the Leap Motion SDK supports only a very small number of recognizable gestures. These include circle, swipe, key tap, and screen tap. From the Leap Motion site, each gesture is described as follows:

Circle:	A single finger tracing a circle.
Swipe:	A linear movement of the hand.
Key Tap:	A tapping movement by a finger as if tapping a keyboard key.
Screen Tap:	A tapping movement by the finger as if tapping a vertical computer screen.

3.2.1b Usefulness

While these included gestures work very well, they are far too limited for any significant application. For example, a “swipe” gesture is simply a “linear movement of the hand”, which is completely independent of direction, velocity, and distance. While this information is accessible, it means that significant alteration from the included gestures would be necessary for any application which has even a moderate need for gesture variety.

More significantly, the included gestures do not contain any complex or multi-part gestures. These would be gestures which could consist of one hand performing multiple small gestures or multiple hands. The latter, in particular, is useful. As an example, a two-handed “pinch-to-zoom” mechanism would be very useful, but is not presently supported in any way.

3.2.2 Tracking Data

The Leap Motion also allows developers to access significant amounts of tracking data from the device. Even though particular gestures are not always recognized, the Leap Motion still tracks hands, fingers, and tools as individual elements. The data reported about each of these includes information about the movement, including direction and velocity. Additionally, it provides information, when available, about the position and orientation of the user's hands.

With this information, it is possible to create a much greater variety of gestures that can recognized by the Leap Motion. This will be necessary for any application which requires more gesture support than those required in the SDK. This would allow us to recognize the complex gestures which would otherwise be unusable.

Chapter 4: System Goals

At a high level, the goal of this project is to create a Software Development Kit which will allow developers to use gesture-recognition devices in programs with limited effort. The system should support reasonably complex gestures, and should be simple to use.

In order to avoid confusion, we name our SDK. For the duration of this paper, the developed SDK shall be referred to as the "Dagger SDK" ("Device AGnostic GEsture Recognition Software Development Kit"), or simply "Dagger".

4.1 Simplicity

Simplicity for developers using the Dagger SDK is one of the most important features to be incorporated. It is vital that this new SDK represent an improvement from the existing SDKs. Since the Dagger SDK will rely on an existing device, and, in turn, its native SDK, it is impossible to truly add any new functionality. By strict definition, if we use the device to create the functionality, then the device already had the potential for that functionality. However, we can greatly increase the simplicity with which this functionality can be achieved. That is, we can use the available technology in a manner as to make advanced and novel functionality accessible.

The Dagger SDK will concentrate on allowing developers to implement features of a program which use gesture recognition, or to add gesture recognition to an existing program. There will be two primary components exposed to users for this functionality. These are the definition or initialization of gestures, and the ability to set up handling for individual gestures.

The code to define gestures must be very simple. Currently, we can find no SDKs which support customizable gestures without requiring the developer to code the entire gesture recognition. Customizability of gestures is vital to Dagger, as many developers would require finer control over what constitutes a gesture than simply using the built-in definitions which may come with a device's SDK. For instance, though the Leap Motion SDK provides a built-in “swipe” gesture, it provides no ability to customize what defines a “swipe”.

Additionally, the ability to set up handling for different gestures must be simple for developers to implement. The Dagger SDK should allow developers to register a particular gesture, and then be notified when this gesture has been performed. In this respect, Dagger would allow gesture recognition to perform in a similar manner to many keyboard handlers: register interest in a given gesture, and handle the action when the gesture occurs.

4.2 Gesture Support

Since gesture recognition is the purpose of the Dagger SDK, it is important to clarify the different types of gestures we wish to support.

4.2.1 Single-Trackable Gestures

Single-Trackable gestures tend to be the simplest gestures which are recognized. These are gestures which can be performed using only a single trackable entity (e.g., a hand or a finger). One example of a single-trackable gesture is the ubiquitous “swipe” gesture, in which a user moves his or her hand in a given direction.

Even though single-trackable gestures are the simplest archetype of the gestures which must be recognized, there is still a significant amount of customization which should be available to developers. For instance, if we further examine the swipe gesture, we might ask whether a “right swipe” should be treated differently than a “left swipe”. Thus, there must be a distinction between these two gestures. Additionally, there is an option that only a right swipe should be recognized, so it is not enough to merely include the angle in any report - we must also be able to filter by angle.

Angle is not the only attribute which we must consider; others include distance, velocity, and duration. Dagger should be able to classify gestures based on all these attributes, and, in order to keep with the goal of simplicity, must not report gestures which do not meet these criteria.

4.2.2 Relational (Multi-Trackable) Gestures

There are also gestures which require multiple trackable objects to perform. One of the most obvious of these is the stereotypical, and now nearly-natural, “zoom” gesture. In this gesture, the user moves two fingers (or hands) away from each other in opposite directions. As the distance between the two trackables increases, the application zooms in; as the distance decreases, the application zooms out.

In this case, we must provide support for a gesture which has multiple components. Each component of the gesture can be represented via a single-trackable gesture; however, this is not sufficient for the entire gesture. There also must be some representation of the relation between the two simple gestures.

This relation must be able to recognize different attributes of the two gestures. For instance, in the case of the zoom gesture, there are two aspects which we are relevant. The first is that the gestures must be moving in opposite directions. The second is that we must recognize what the distance between the two gestures is.

It is not enough to simply register multiple simple gestures as a substitute for a relational gesture. Even in the simple case of the zoom gesture, we find that we would need to register a swipe gesture and compare each performance of the gesture to determine if another gesture happened at the same time, and in the opposite direction. This is too cumbersome for our purposes.

4.2.3 Multi-Stage Gestures

The final type of gestures which we should recognize are multi-stage gestures. This is a gesture which is immediately followed by another gesture. An example of this might be a “grab and drop” gesture, in which the user “grabs” an object, moves it, and drops it. In this case, there are a series of simple gestures - grab, move, and drop - which are performed in order. Following the performance of the final simple gesture ("drop"), the gesture is completed.

These gestures are more feasible to complete via a collection of simple gestures. However, it is still cumbersome to do so, and should not be required of developers, in order to keep in line with the goal of simplicity.

4.2.4 Continuous/Updating Gestures

The system should also support the concept of continuous gestures. These are gestures which meet a minimum requirement, but continue to have an effect for so long as they remain valid. This is a very common desire in order to allow for providing the user with immediate feedback, and supporting a more polished, gradual approach.

An example application of this can again be seen with the “zoom” gesture. If the user wishes to zoom in, there may be a small minimum distance which must be met, after which the application will begin to zoom. However, the zoom gesture should not happen in one burst at the culmination of the gesture, and nor should it consist of a number of small updates each time after the minimum distance is met. Instead, once the minimum distance is met, the application should zoom as the user continues to perform the gesture (by moving his or her fingers farther apart or closer together).

In order to satisfy the need for this type of gesture, the system must provide a mechanism for an updating gesture, which has completed the minimum requirements, but is not completed. The system should notify a listener with each subsequent update following the initial match of the gesture.

4.3 Device Abstraction

The Dagger SDK has been developed using a particular device; however, it should, to the extent possible, be device-agnostic. This means that the vast majority of the Dagger should be device-independent, and an additional layer will be added to adapt the Dagger

SDK to a particular device. There are multiple reasons for this abstraction, which are discussed below.

4.3.1 Use with Multiple Devices

The most obvious reason for creating an SDK which is largely agnostic towards the device used is to allow developers to use different devices. Though currently there are few devices which would meet the requirements for use with the SDK, this is likely to change. The Microsoft Kinect is being replaced by the Kinect One, Apple has recently acquired Prime Sense, and new devices are also on the horizon. If Dagger were to be firmly fixed to a single device, it is far less useful, and will likely suffer from a very short-lived existence. Allowing developers to use it with multiple devices will encourage growth and adoption.

4.3.2 Use with Multiple Existing SDKs

Naturally, it is impossible to create a truly universal SDK which is “plug-and-play” for all devices. Some sort of layer will be necessary, and, towards this end, another SDK must be utilized. Whether this is the device's native SDK or a third-party or open-source alternative, there must be an additional layer between our the Dagger SDK and the device itself. Abstracting out the device as much as possible inherently also abstracts out the other SDK used with the device. This allows developers more choice, and the power to experiment.

4.3.3 Ease of Updating

As a device-independent SDK, the updating process will be far easier for Dagger. Since there will be a devoted layer to each device, only that layer must be updated when the device is updated, rather than the entire project. For one, this makes the updating process easier in code - as all the code which may need to be revised is in one section. Another advantage is that anything which breaks in a device update will be concentrated to that particular layer - other layers, and the Dagger core, will still work.

Chapter 5: Development Device

Though the Dagger SDK should be device-agnostic, we must develop it using a particular device in order to test the validity of the developed product. For this reason, we must determine which of the available products is most appropriate for development.

This project will primarily concentrate on the performance of smaller gestures, such as those performed by a user's fingers and hands. This is because these gestures are typically more suited to a typical environment in which one uses a computer, and should hopefully reach a wider audience. While the larger gestures, those performed by an entire limb or full body, do have a large market in entertainment and a number of other markets, we feel they are not as suited to most existing applications, in which users are typically confined to a relatively small area, e.g. sitting at a desk.

The two devices which are likely candidates for this project are the Leap Motion and the Microsoft Kinect, which are the two most popular (and commercially available) camera-based gesture recognition devices. Because of the focus on these smaller gestures, the device which is most appropriate is the Leap Motion. Through previous experimentation in a Human-Computer Interaction course, we have found that the Leap Motion is more accurate than the Microsoft Kinect in the domain of small gestures.

For this reason, we have decided on the use of the Leap Motion in the development of the Dagger SDK.

Chapter 6: System Design

The following section describes, in detail, the different workings of the system.

6.1 System Overview

This novel system has three main components - gestures, patterns, and the tracking system itself. “Patterns” are patterns which the developer registers with the system, and are then matched. “Gestures” are completed actions which match the registered patterns. The “system” component is responsible for the actual observation of gestures, and reporting any observed gesture to the registered listener.

We can also view these in terminology borrowed from Russel and Norvig's *Artificial Intelligence: A Modern Approach*. Here, a "Pattern" would be a collection of certain percepts, which form an episode. The "Gestures" are the user's own performance of these episodes. Finally, the "system" completes the mapping between episodes and programmatic actions.

Throughout this paper, we will use the terms "Pattern" and "Gesture" to refer to these concepts.

6.2 Patterns

Patterns are at the lowest level, and are unaware about either the system or gestures. These represent the movements which the user must perform in order to complete a gesture. A pattern is specified by the developer, and then matched by the system.

A Pattern should be able to describe a vast number of possible motions which the developer wishes to handle. A number of examples include "swipes" (in any direction, with any speed, and so on), "zoom" patterns (either single- or multi-handed), basic hand signals, pointing, and others. Examples of each class of Pattern will be described in the correlating section.

The Patterns package is independent of the Gestures package and the System package, requiring no knowledge from and having no dependency upon either.

6.2.1 Attributes

The “building blocks” of Patterns are a collection of attributes. The attributes are as follows:

6.2.1a Basic Attributes

Distance: the total distance which was traveled over the course of the entire gesture, independent of angle, velocity, or any other factor.

Time: the total amount of time which has elapsed in the gesture, measured in microseconds.

6.2.1b Angle Attributes (degrees)

Initial Angle: the angle at which the gesture began, measured in degrees.

Final Angle: the final angle of the gesture.

Average Angle: the average angle throughout the performance of the gesture.

Absolute Angle Change: the gross amount that the user changed the direction during the gesture; thus, if the user changes direction from left to up, and then stays up, the total angle change is 90-degrees. If the user were to then switch back to left, then the absolute angle change is 180-degrees, even though there is no net change.

Maximum Angle Change: the maximum amount that the user changed from the original path; this is useful if the developer is not particular about the steadiness of the user's hand, but wants to know whether or not the user's direction is roughly the same.

6.2.1c Velocity Attributes (millimeters/microsecond)

Initial Velocity: the initial velocity of the gesture.

Final Velocity: the final velocity of the gesture.

Average Velocity: the average velocity throughout the gesture.

Minimum Velocity: the minimum velocity which occurred in the gesture.

Maximum Velocity: the maximum velocity which occurred in the gesture.

6.2.1d Sub-Trackable Count Attributes

Initial Sub-Trackable Count: the initial number of sub-trackables associated with the primary trackable in the gesture.

Final Sub-Trackable Count: the final number of sub-trackables in the gesture.

Minimum Sub-Trackable Count: the minimum number of sub-trackables in the gesture.

Maximum Sub-Trackable Count: the maximum number of sub-trackables in the gesture.

6.2.1e Attribute Selection

Naturally, there are a large number of possible attributes. The attributes above were selected in order to allow for a large variety of significantly-different gestures to be performed, while still allowing for strict determination of acceptance.

These attributes were also selected for the intention of keeping the Dagger SDK simple, but also powerful. If the attributes became too specific, two problems could arise. First, Gestures could become exceedingly difficult to perform, as too many complex factors are required. Second, it would greatly hinder the efforts to make this SDK available for multiple devices. These attributes are relatively straight-forward, and do not require complicated information - rather, they rely on the position and number of trackable objects. Other attributes would require extensive additional work, inhibiting the expansion of Dagger.

6.2.2 PartialPattern

A PartialPattern is essentially a collection of requirements for different attributes. Developers can register maximum, minimum, or exact requirements. A PartialPattern is matched when all the requirements have been met. For example, we may look at a generic “Swipe” PartialPattern, in which the user simply moves his or her hand in a consistent direction, e.g., to move something on the screen. This pattern may have the following requirements:

Distance (Minimum): 200

Maximum Angle Change (Maximum): 30

Here, we specify a pattern which has a minimum distance of 200 (mm) and a maximum angle change of 30 (degrees). This means that the user must move his or her hand at least 200 mm in a direction without moving more than 30 degrees from the original path of travel. However, there is no initial or final angle requirement, so the user's hand may move in any direction, as long as it never varies more than 30 degrees from the initial angle.

Also note that the ability to register minima and maxima with the pattern is not superfluous with the minimum and maximum attributes, even though it, at first glance, may seem so. For instance, by registering a minimum “minimum velocity”, the developer ensures that, even at the slowest speed, the gesture was above a certain threshold.

6.2.3 SimplePattern

A SimplePattern represents the simplest full pattern. It consists only of a single PartialPattern, as that is all that is necessary for it to be completed. This means that examples of SimplePatterns include swipes, basic hand signals (such as "stop"), or an open-to-closed hand movement. The SimplePattern is matched any time the associated PartialPattern is performed.

6.2.4 RelationalPattern

A RelationalPattern represents a pattern which requires two PartialPatterns, as well as a given relation to be met. The relation between the two PartialPatterns is specified by a different set of attributes:

Angle Diference:	The difference in angle between the two PartialPatterns.
Distance Difference:	The difference in distance between the two PartialPatterns.
Start Position Difference:	The difference in the starting position between the two PartialPatterns.
End Position Difference:	The difference in ending position between the two PartialPatterns.

These relations also have associated minimums and maximums. Using these, we could define a common “zoom” gesture as having two “swipe” gestures (from the previous section), in addition to the relation which has an angle difference between 160 and 200 degrees (meaning that the two swipes are moving in roughly opposite directions).

6.2.5 Two-PartialPattern Limit

The question of why there are only two PartialPatterns allowed in a RelationalPattern is immediately prompted. The reason is that the extra computation, as well as the extra code necessary on the part of the developer using the Dagger SDK, necessary for a RelationalPattern with three or more PartialPatterns is simply too excessive. It would require for an additional PartialPattern to match and check, as well as requiring for a more complex Relation system, which would greatly reduce the simplicity with which Relational patterns could be registered.

Additionally, the number of gestures performed with more than two trackable objects is very limited - since users only have two hands, and only very coordinated users would be able to move three fingers in three distinctly different manners.

6.2.6 MultiStagePatterns

MultiStagePatterns represent patterns which are performed in sequence. That is, the first PartialPattern is performed immediately prior to the second PartialPattern. An example of a MultiStagePattern would be a grab-and-drop gesture. These gestures are actually very straight-forward - they are simply a collection of PartialPatterns to be performed.

Unlike with RelationalPatterns, there is no limitation on the number of PartialPatterns associated with a MultiStagePattern. However, the full MultiStagePattern must be performed by a single trackable object. This is due to the desire to not encourage developers to use MultiStagePatterns as state-machine helpers, or to chain multiple complex gestures. The purpose of each MultiStagePattern is to represent a single pattern, which happens to require multiple small steps - rather than to represent a chain of full, distinct patterns, which should be logically separate.

6.3 Gestures

Gestures are, in essence, the actual performance of a pattern by the user. Instead of having a collection of requirements, they have actual values, since they represent the true actions of the user. The Gesture package is independent from the System package, but based upon the Patterns package.

6.3.1 PartialGesture, SimpleGesture, RelationalGesture, MultiStageGesture

PartialGestures, SimpleGestures, RelationalGestures, and MultiStageGestures are directly correlated with their Pattern counterparts. That is a PartialGesture is the performance of a given PartialPattern, a SimpleGesture is the performance of a given SimplePattern, and so forth. There is a PartialGesture for each PartialPattern in the original Pattern. Thus, a RelationalGesture has two PartialGestures - one associated with each PartialPattern in the RelationalPattern.

Each of these PartialGestures has the full suite of attributes associated with the performance, which the developers using the Dagger SDK may use if desired. For instance, the developer may wish to treat a swipe of a given velocity or distance differently.

6.3.1a Why Include All Attributes?

The question of why to include all attributes with all gestures also arises. Since the developer registers a collection of requirements with the creation of the Pattern, the system could feasibly ignore all attributes which are not included amongst the registered requirements.

While this is true, there is no guarantee that the developer does not want access to attributes which do not have an associated requirement. For instance, the developer may classify any straight movement in a given direction as a swipe, independent of velocity, and yet may also want to treat a fast movement as more severe than a slow movement.

Additionally, the attributes were designed to be quick to calculate; thus, the comparison in order to determine whether or not to store the attribute is almost as expensive as the calculation and storage itself.

6.3.2 InProgressGestures

An InProgressGesture represents a partially-performed PartialPattern. Each InProgressGesture has a single associated PartialPattern, which it strives to match. As the gesture is continually performed, the InProgressGesture continues to update attributes. The InProgressGesture is updated at every frame, and stores these frames for later use. The InProgressGesture also has a given MatchStatus, which is one of the following:

Match:	The gesture is a match to the pattern; all requirements have been met.
Possible:	The gesture does not match the pattern yet, but may do so if given more movement to update. For instance, the gesture may need to cover more distance in order to be completed.
Impossible:	The gesture does not match the pattern, and can never do so. This is the case when a hard limit, such as distance maximum, is exceeded.

For so long as the InProgressGesture is a Match or is Possible, it only stores and updates all the frames pertaining to it. However, if the InProgressGesture is ever determined to be an Impossible match, then the InProgressGesture will begin removing frames (earliest frame removed first) until it is at least a Possible match once again.

In this way, InProgressGestures are consistently kept valid - they will continually try to match a PartialPattern.

6.4 System

The “system” contains those components of the Dagger SDK which relate directly to the tracking of objects and recognition of performed patterns. Though Dagger has been designed to partition the work such that the high-level system does not need to directly match patterns to movements (this is taken care of by the InProgressGestures and their underlying Patterns), the system is the portion which ties all the pieces together, and informs a listener when a Gesture has been performed. The System package relies upon both the Patterns and the Gestures packages.

6.4.1 TrackableInfo

Each trackable object (e.g., hands or fingers) has an associated TrackableInfo. The TrackableInfo also has a list of all PartialPatterns which may be associated with the given trackable object; that is, if the TrackableInfo is associated with a hand, it will have all PartialPatterns which could be performed by a hand.

There is a mapping between PartialPatterns and InProgressGestures. Since InProgressGestures automatically update to try and fit a given PartialPattern, this mapping serves to continually try and match all possible patterns which apply to the trackable.

Trackables are given a particular unique ID by which they are referenced. Since trackable objects may sometimes be “lost”, there must be a lifespan of TrackableInfos in order to ensure that TrackableInfos from lost tracked objects do not persist indefinitely. Towards this end, each TrackableInfo has a life amount. If this life amount is depleted, the TrackableInfo is marked as invalid. The life amount is refreshed every time that the TrackableInfo is updated, since this clearly indicates the tracked object is still present.

6.4.2 TrackingGroup

A TrackingGroup is a collection of TrackableInfos for a given type of trackable object. Thus, there would be one TrackingGroup for hands, another TrackingGroup for fingers, and so forth. This group contains a TrackableInfo for each trackable object of the given type, and will update all the TrackableInfos with the information from a given frame.

The TrackingGroup is also responsible for decrementing the life of unused TrackableInfos after a frame update. This is necessary in order to ensure that lost entries are eventually removed. A final responsibility of the TrackingGroup is to indicate which TrackableInfos have been “consumed” for the use of a particular Gesture. It is vital that, even though a tracked object's movements may meet the requirements for multiple Patterns, the movement is only consumed once, and is never double-counted.

6.4.3 TrackingSystem

The TrackingSystem is the object which owns all the TrackingGroups, and updates them at every frame. After this update, the TrackingSystem will check for any completed Gestures, and if any exist, will notify a listener. A very rough pseudo-code implementation of the algorithm is available in the appendix.

The TrackingSystem is also the object with which a listener, and thus the developers using the Dagger SDK, will interact. It is responsible for registering all Patterns, and assigning the proper PartialPatterns to the appropriate TrackingGroups. Additionally, the TrackingSystem must maintain all Gestures which are currently updating, and notify the listener of these, as well.

6.5 Listening

We implement a listener design pattern in order to be notified of when a gesture is performed.

6.5.1 TrackingSystem.Listener

There is a listener for the TrackingSystem, which client objects implement. The TrackingSystem notifies this listener any time a gesture is completed, updated, or terminated. The listener provides the system with feedback regarding whether or not the gesture should continue updating.

6.5.1a CompletionStatus

There are two possible completion statuses which the listener may report to the TrackingSystem.

Complete:	The gesture is fully complete, and should not continue to be updated.
Incomplete:	The gesture has matched, but may not be fully complete. The listener is actively processing the gesture, and this handling has a fluid termination, meaning that the listener may require information about the gesture for so long as it remains valid. For instance, if the user wishes to zoom, the gesture should be fluid, and continuously updated.

6.5.1b onGestureComplete

The listener's onGestureComplete() method is called when a Pattern is initially matched. That is, there is a minimally-satisfactory Gesture for the Pattern. The listener is informed of the Gesture, and has the option of reporting back that the Gesture is complete (CompletionStatus.Complete), or that it is continuing (CompletionStatus.Incomplete).

If the listener reports that the Gesture has been completed, the system will terminate all information about the performed Gesture (the underlying Pattern, of course, remains). The System will begin again matching the Gesture from scratch, and any future reports will be from newly-performed Gestures.

If the listener instead reports that the Gesture is incomplete, the TrackingSystem will continue to update the Gesture. At every frame, the TrackingSystem will notify the listener of any updates to the existing Gesture via the onGestureUpdated() method.

6.5.1c onGestureUpdated

Calling the onGestureUpdated() method signifies that a previously-completed Gesture has been updated, and remains valid. This will only be called if the Gesture was previously reported via onGestureComplete() and the listener returned an “Incomplete” CompletionStatus.

Similar to onGestureComplete(), the listener may again return either CompletionStatus, and the results are the same. If a Complete status is returned, then the Gesture will cease to be updated; if an Incomplete status is returned, the Gesture will continue to be updated (and the listener notified) for so long as the Gesture remains valid with the underlying Pattern.

6.5.1d onGestureTerminated

The onGestureTerminated() method is called when an updating Gesture is no longer valid, and will not continue to update. It is sent to notify the listener that any in-progress (and non-atomic) functions dependent on the Gesture should be ended. For instance, if the Gesture correlates to a drag-and-drop action, and the drag ceases to be valid, then the drop should commence at the current location, rather than silently ceasing to update the action.

The onGestureTerminated() method will only be called with a previously-updating Gesture, and will not be reported when the listener deliberately terminates a Gesture, as in by returning a Complete status from onGestureUpdated().

6.6 Leap Motion Layer

The Leap Motion layer is not a part of the core Dagger SDK; rather, it is necessary to connect the Leap Motion device for use with Dagger. Since Dagger is designed to be mostly device-agnostic, there must be an intermediary layer. This layer is responsible for converting the Leap Motion's data into the format which is expected by the TrackingSystem. This layer can be used by developers using Dagger, or they may substitute their own (either for the Leap Motion device, or for another device).

6.7 Notes on Device Abstraction

While device abstraction remains a primary goal of the Dagger SDK, it is out of the scope of this project for Dagger to be entirely device-abstract, and rely solely on the raw data from the device. For one, this would still require some sort of standardized data format, or conversions from each of the possible formats. Additionally, this would require the equivalent of the complete redesign of a device's SDK.

Instead, Dagger requires input from an intermediary layer (e.g., the Leap Motion layer mentioned above). This layer provides the locations of the tracked objects. With the Leap Motion, this is relatively simple using the built-in SDK. With other devices, this may require more intermediary work, especially if working from scratch (i.e., the raw camera feed). However, as mentioned while discussing attribute selection, Dagger was designed to require simple data, rather than advanced tracking information.

Chapter 7: System Requirements

In order to verify that the goals of the system, outlined in a previous section, are met, we must define explicit pass-fail requirements for each of these goals. These requirements are outlined below and evaluated below.

The definitions for the requirements of the system in respect to each of the three primary goals are below.

7.1 Simplicity

1. A new PartialPattern with any number of attributes can be defined in a single statement of code.
2. Given existing PartialPatterns, a new SimplePattern can be defined in a single statement of code, with no nested instantiations.
3. Given existing PartialPatterns, a new RelationalPattern can be defined in a single statement of code, with at most one nested instantiation.
4. Given existing PartialPatterns, a new MultiStagePattern can be defined in a statement of code, with at most one nested instantiation.
5. Given an existing Pattern and TrackingSystem, the Pattern can be registered in the system in a single statement of code.

7.2 Gesture Support

1. The system shall recognize a single-trackable gesture.
2. The system shall recognize a relational gesture.
3. The system shall recognize a multi-stage gesture.

4. The system shall allow developers to treat a gesture as terminal, in which the gesture is fully completed after it is first matched to a pattern.
5. The system shall allow developers to treat a gesture as continuous, in which the gesture is continuously updated after it is first to a pattern.
6. The system shall allow developers to terminate a continuous gesture at any update.
7. The system shall allow developers to handle terminated gestures (i.e., gestures which were previously continuous, but are no longer valid).

7.3 Device Abstraction

1. The core of the system shall not have dependencies upon nor utilize methods from any device's native SDK.

Chapter 8: System Use and Validation

This section outlines sample uses of the system in order to illustrate its key points of simplicity and powerful gesture recognition. Additionally, this section serves as a form of validation for the requirements defined in the previous section. Unfortunately, this is not complete validation - for true validation of the system, it will have to be used by a number of outside developers, ideally on multiple devices. However, this is out of the scope of this project.

8.1 Requirements Evaluation

In this section, we evaluate each of the requirements from Chapter 7.

8.1.1 Simplicity 1

Below is the code to instantiate a PartialPattern for a "swipe" movement. This particular movement requires that the user move a given trackable object for a minimum distance of 120.0 (mm), and deviate from the original path of travel by no more than 20 degrees. An arbitrary amount of additional attribuets may be registered; additional examples include registering a minimum speed, a maximum absolute angle change, or maximum distance.

```
PartialPattern swipePartial =
    new PartialPattern("Swipe Partial")
        .registerMin(Attribute.DISTANCE, 120.f)
        .registerMax(Attribute.MAXIMUM_ANGLE_CHANGE, 20.f)
        [.register[Min,Max,Exact](Attribute.<Name>, <value>)];
```

The code is contained within a single statement, thus satisfying the requirement.

8.1.2 Simplicity 2

The code below instantiates a SimplePattern for a “swipe” movement, relying upon the declaration of the swipe PartialPattern from the previous example. In this example, the pattern is specified to target a hand movement.

```
SimplePattern swipePattern =
    new SimplePattern(“Swipe Pattern”,
                      Trackable.HAND,
                      swipePartial);
```

The code is contained within a single statement with no nested instantiations, thus satisfying the requirement.

8.1.3 Simplicity 3

The following is the code to instantiate a RelationalPattern for a stereotypical “zoom” gesture, in which the user's fingers move apart from one another to zoom in, or move towards one another to zoom out. This relies upon the swipe PartialPattern from the previous example, and creates a relation between the two simple patterns in which the angle between the directions of the two swipe patterns must be between 160 and 200 degrees (roughly opposite, with exactly opposite at 180 degrees).

```
RelationalPattern zoomPattern = new RelationalPattern(
    "Zoom Pattern",
    Trackable.FINGER,
    swipePartial,
    swipePartial,
    new Relation(Relation.Type.ANGLE_DIFFERENCE, 160.f, 200.f));
```

The code is contained within a single statment with a single nested instantiation for the relationship between the two PartialPatterns, thus satisfying the requirement.

8.1.4 Simplicity 4

This is the code to instantiate a “grab and drop” pattern. In this, the user makes a grabbing motion at an object, then moves his or her hand (“holding” the object while doing so), and drops it in a new location, analogous to the drag-and-drop of a computer mouse. The “grab” partial pattern consists of starting with five countable sub-trackables - that is, the five fingers of an open hand - and culminating in no sub-trackables - no fingers, since the user has “grabbed”. The “move” partial pattern lasts for so long as the user's hand does not have any visible sub-trackables - once sub-trackables re-emerge, the user has “dropped” the object, and the pattern will be terminated by the system (since the updating pattern is no longer valid). In order to reduce strictness, a maximum number of sub-trackables could be registered for the “move”, rather than a hard limit of zero.

```
PartialPattern grab = new PartialPattern("Grab")
    .registerExact(Attribute.INITIAL_SUBTRACKABLE_COUNT, 5)
    .registerExact(Attribute.FINAL_SUBTRACKABLE_COUNT, 0);
PartialPattern  grabbingMove  =  new  PartialPattern("Grabbing
Move")
    .registerExact(Attribute.INITIAL_SUBTRACKABLE_COUNT, 0)
    .registerExact(Attribute.FINAL_SUBTRACKABLE_COUNT, 0);
MultiStagePattern grabAndDropPattern = new MultiStagePattern(
    "Grab and Move Pattern",
    Trackable.HAND,
    new PartialPattern[] { grab, grabbingMove });
```

The code for the instantiation of the MultiStagePattern is contained within a single statement with a single nested instantiation for the array of PartialPatterns, thus satisfying the requirement.

8.1.5 Simplicity 5

This shows the simple code necessary to instatiate a new TrackingSystem, register a listener (presumed to exist), and register a pattern (borrowed from the previous example).

```
TrackingSystem trackingSystem = new TrackingSystem();
trackingSystem.registerListener(listener);
trackingSystem.registerPattern(grabAndDropPattern);
```

The code to register the pattern with the system is done in a single compiled line of code, thus satisfying the requirement.

8.1.6 Gesture Support 1 - 3

These requirements specified that the system must recognize the three types of gestures - Simple, Multi-Trackable (Relational), and Multi-Stage. These requirements have been met, but cannot be shown textually.

8.1.7 Gesture Support 4

This is the example code for a simple onGestureComplete() implementation of a TrackingSystem listener in order to indicate that the gesture is terminal, and should not be updated. For instance, a terminal swipe will be reported only once, even if the swipe continues after the initial report (unless the swipe continues long enough to be considered an independent and complete gesture). By returning CompletionStatus.COMPLETE, we indicate that the gesture is complete, and should be terminated.

```
public CompletionStatus onGestureComplete(Gesture gesture) {
  handleGesture(gesture);
  return CompletionStatus.COMPLETE;
}
```

The system allows for a completed gesture to be treated as terminal, thus satisfying the requirement.

8.1.8 Gesture Support 5

Similar to the previous example, this demonstrates an implementation which will inform the TrackingSystem that the gesture is not yet complete, and should be updated. This will allow the listener to respond to an the gesture changes at every frame. For instance, a swipe which is incomplete (or “continuous”) will continue to send “updated” calls to the listener. Only one call will be placed to onGestureComplete, in order to avoid confusion with other trackable objects which may complete the same pattern. In this particular case, the gesture updates until invalid.

```
public CompletionStatus onGestureComplete(Gesture gesture) {
  handleGesture(gesture);
  return CompletionStatus.INCOMPLETE;
}

public CompletionStatus onGestureUpdated(Gesture gesture) {
  handleGestureUpdate(gesture);
  return CompletionStatus.INCOMPLETE;
}
```

The system allows for a completed gesture to be treated as continous, thus satisfying the requirement.

8.1.9 Gesture Support 6

In this example, we initially inform the TrackingSystem that the gesture should be updated, but later decide to terminate the gesture. This prevents future updates from the TrackingSystem for this particular gesture, and allows the for the use of the trackable object in other patterns. In this example, we assume that there is a litmus test, in the form of shouldTerminateGesture(), for whether or not the developer desires to terminate the gesture (e.g., perhaps the action the gesture incurs has reached a maximum, and further performance is irrelevant).

```
public CompletionStatus onGestureComplete(Gesture gesture) {
  handleGesture(gesture);
  return CompletionStatus.INCOMPLETE;
}

public CompletionStatus onGestureUpdated(Gesture gesture) {
  handleGestureUpdate(gesture);
  return shouldTerminateGesture(gesture) ?
      CompletionStatus.COMPLETE :
      CompletionStatus.INCOMPLETE;
}
```

The system allows developers to terminate a continuous gesture at any update, thus satisfying the requirement.

8.1.10 Gesture Support 7

The (admittedly simplistic) code below describes how a developer would handle a gesture which was terminated, i.e. was continuous, but became invalid. This is a powerful tool, and can be necessary in some patterns. For instance, in the grab-and-drop pattern described above, the gesture relies upon the call to onGestureTerminated() in order to inform the developer that the object should be "dropped".

```
public void onGestureTerminated(Gesture gesture) {
  handleGestureTerminated(gesture);
}
```

The system allows developers to handle terminated gestures, thus satisfying the requirement.

8.1.11 Device Abstraction 1

This requirement is not demonstrable through code; however, it can be seen in the UML and class diagrams in the appendix that the core of the gesture recognition system has no reliance upon the Leap Motion layer. Since this layer is the only portion of the Dagger SDK which references the Leap Motion's own SDK, this requirement is satisfied.

8.2 Comparison to Current SDK

It is important to compare the Dagger SDK with the existing SDK. If the development of Dagger was successful, then there should be significant improvement over the official

SDK. Towards this end, we examine how each SDK would handle different types of gestures, and address the issue of device abstraction.

8.2.1 Single-Trackable Gestures

First, we examine the case of single-trackable (simple) gestures. The official SDK for the Leap Motion supports four classes of these gestures, whereas the developed SDK provides the ability to customize a gesture requirement. This allows the developer to much more easily specify handling for a particular case of gesture, since each customization can be treated differently.

We can look at the case of a left-swipe as an example. In the official Leap Motion SDK, we can only specify that we should recognize “swipes” in general; we cannot specify that a swipe must be left-moving. While we can discern whether or not the swipe moved to the left after the swipe is reported, it is something which must be done in every case. Similarly, checks must be made for any other requirements, such as distance, time, or other desired constraints.

This results in significant extra effort on the part of the developer to determine how to handle a given swipe, or even if a given swipe should be handled. The pseudo-code for this would be:

```
controller.enableGesture(Gesture.Type.TYPE_SWIPE);
...
onFrame(Frame frame) {
  for (Gesture gesture : frame.gestures()) {
    if (gesture.type()) == TYPE_SWIPE) {
      SwipeGesture swipe = new SwipeGesture(gesture);
             if  (swipe.position()  -  swipe.startPosition()  >=
minDistance
          [&& meetsOtherRequirements(swipe)]) {
        // no option for angle tolerance
        handleSwipeGesture(swipe);
      }
    }
  }
}
```

While this is relatively straight-forward and not an exorbitant amount of code for this particular gesture, this type of verification must be done for every single gesture specified by the developer.[2] In contrast, the pseudo-code while using the Dagger SDK is:

[2] the Leap Motion SDK does allow for a "minimum distance" and "minimum velocity" requirement upon gestures reported; however, these cannot be changed for different gestures. Thus, if the developer wants one type of swipe to have a different minimum velocity than another, he or she must still verify by hand.

```
PartialPattern swipePartial = new PartialPattern("swipe")
    .registerMin(Attribute.DISTANCE, 120.f)
    .registerMax(Attribute.MAXIMUM_ANGLE_CHANGE, 20.f)
    [.register[Min,Max,Exact](Attribute.<Name>, <value>)];
SimplePattern swipePattern =
    new SimplePattern("Swipe", Trackable.HAND, swipePartial);
system.registerPattern(swipePattern);
...

onGestureComplete(Gesture gesture) {
  if (gesture.pattern() == swipePattern)
    handleSwipeGesture(gesture);
}
```

This code holds for any pattern, no matter the complexity of the gesture.

Further, the native SDK does not filter out any gestures which match the gesture archetype. That is, there will be many more false positives reported. In the example above, all swipes (whether they meet any of the additional criteria or not) will be reported. With the Dagger SDK, the only gestures reported will be those which the developer explicitly specified.

8.2.2 Multi-Trackable Gestures

With multi-trackable (relational) gestures, the official SDK offers no graceful mechanism.

For one thing, relational gestures depend upon simple gestures - thus, all complexities from the simple gestures are inherited for relational gestures.

Further, each simple gesture is reported by the official SDK individually, meaning that the developer will have to keep a record of all reported gestures, and then determine whether or not two of the gestures reported match the necessary simple gestures, determine whether or not their relation is satisfactory, and determine whether or not the two gestures occurred in an appropriate time window (since they should be performed simultaneously).

Each of these determinations is somewhat complex; thus, including the pseudo code for the implementation using the official SDK is cumbersome.

```
controller.enableGesture(Gesture.Type.TYPE_SWIPE);

onFrame(Frame frame) {
  ArrayList<SwipeGesture> swipes =
      new ArrayList<SwipeGesture>();
  for (Gesture gesture : frame.gestures()) {
    if (gesture.type()) == TYPE_SWIPE) {
      SwipeGesture swipe = new SwipeGesture(gesture);
      if (swipe.position() - swipe.startPosition() >=
              minDistance) {
        // no option for angle tolerance
        swipes.add(swipe);
      }
    }
  }

  for (int i = 0; i < swipes.size() - 1; ++i) {
    for (int j = i + 1; j < swipes.size(); ++j) {
      if (areOpposite(swipes.get(i).direction(),
                      swipes.get(j).direction()) {
        handleZoomGesture(swipes.get(i), swipes.get(j));
        i += 1;
        break;
      }
    }
  }
}
```

Even after all this, developers would still have to do a number of other tasks. These include keeping track of consumed gestures, in order to ensure that no gesture is used in more than one multi-trackable gestures; and keeping track of continuous (updating) gestures, since they are not always reported. Further, the amount of customization for these gestures is limited.

However, the pseudo-code using Dagger is perfectly analogous to the same as the previous example:

```
PartialPattern swipePartial = new PartialPattern(“swipe”)
    .registerMin(Attribute.DISTANCE, 120.f)
    .registerMax(Attribute.MAXIMUM_ANGLE_CHANGE, 20.f);
RelationalPattern zoomPattern = new RelationalPattern(
    “Zoom”, Trackable.FINGER, swipePartial, swipePartial,
    new Relation(Relation.Type.ANGLE_DIFFERENCE, 160.f, 200.f));
system.registerPattern(zoomPattern);

onGestureComplete(Gesture gesture) {
  if (gesture.pattern() == zoomPattern)
    handleZoomGesture(gesture);
}
```

In this way, the Dagger SDK is far superior to the official SDK.

8.2.3 Multi-Stage Gestures

Multi-stage gestures, similar to multi-trackable gestures, are not handled well by the official SDK, as there is no support for their concept. Thus, this, too, would have to be checked by hand in the case of every gesture.

This involves multiple checks. First, the developer would need to check whether or not the first stage of the gesture has been met - which inherits all the complexities from the simple gestures. Second, the developer must determine if, once the first stage ends, the second stage has begun, or could potentially begin. Third, the developer must determine that, once the next stage begins, it is completed by the same trackable object. Fourth, the burden of determining when the gesture is terminated falls onto the developer as well.

These determinations are far too complex to include the official SDK's implementation; even a partial implementation is more than a hundred lines of code. As with previous examples, however, the pseudo-code for the Dagger SDK remains simple:

```
PartialPattern grabPartial = new PartialPattern("grab")
    .registerExact(Attribute.INITIAL_SUBTRACKABLE_COUNT, 5)
    .registerExact(Attribute.FINAL_SUBTRACKABLE_COUNT, 0);
PartialPattern   grabbingMove   =   new   PartialPattern("grabbing
move")
    .registerExact(Attribute.INITIAL_SUBTRACKABLE_COUNT, 0)
    .registerExact(Attribute.FINAL_SUBTRACKABLE_COUNT, 0);
MultiStagePattern grabAndDropPattern = new MultiStagePattern(
    "Grab and Move",
    Trackable.HAND,
    new PartialPattern[] { grab, grabbingMove });

<register, handle>
```

Additionally, Dagger allows for the developer to easily terminate a stage of the gesture, while allowing others to continue (by returning either a COMPLETE or INCOMPLETE MatchStatus, respectively). This is something which would again add complexity to an implementation using the official SDK.

8.2.4 Continuous Gestures

The Leap Motion SDK does allow for continuous gestures, but with certain patterns only. Additionally, the support for continuous gestures requires the developer to maintain records of all gestures which are to be referenced, and they must be manually checked. Further, on each of the manual checks, all previous checks for validity must again occur, inheritinig the complexity of all previous sections. Finally, the official SDK does not allow for the termination of gestures. This raises more issues if a gesture should be separated, but is not. Any gestures which could use that information from the continuous gesture must either check it manually, or will not be able to use it.

With the Dagger SDK, the developer may indicate whether or not a gesture should be regarded as continuous or should be terminated by a simple return statement (examples of this are shown in the previous section on requirement evaluation). The developer need not retain any copies of the gestures, as this is handled by the system. The system, naturally, also performs checks for validity. Further, if the gesture is declared invalid (due to a pattern break rather than a developer-signalled termination), the developer will be notified immediately through the onGestureTerminated() method call to the listener.

In this way, again, we see a significant improvement using the Dagger SDK over the official SDK.

8.2.5 Device Abstraction

Clearly, the official Leap Motion SDK is not device-agnostic, and has no intention of ever becoming so. The Dagger SDK has no device dependencies in the core system, and instead provides a layer to connect the device to Dagger. This will allow for multiple devices to be used with the Dagger SDK, provided the appropriate tracking information.

8.3 NASA WorldWind

The NASA WorldWind project is an open-source geographic information system, used both independently and as a basis for other applications. It is very large, at several hundred files, and was not designed for use with this type of 3-D gesture control. As such, it is a very useful test for the real-world applicability and usefulness of the Dagger SDK.

For this, we incorporated a number of different gestures. The first of these gestures is a swipe gesture to pan the globe an amount based upon the velocity of the swipe. The second is a continuously-updating zoom gesture, which is a multi-trackable gesture. The final is a multi-stage (and continuous) grab-and-drop gesture.

All of these gestures function as desired, showing that the Dagger SDK can be used with other applications. The full Dagger-related code for this project is included in the appendix.

8.4 Validation

With this, all requirements outlined for the system are satisfied. This serves as an initial validation of the system, though further validation is beneficial. Ideas for this are outlined in the following section, with other future work.

Chapter 9: Future Work

As with any project, there is an abundance of useful work which could yet be performed on this project. The exact future of the Dagger SDK, and whether or not it will be open-sourced, is currently undecided.

9.1 Integration and Adoption

The primary reason for the development of the Dagger SDK is to allow developers to use it within their own programs to implement gesture recognition. As such, the first step is for this to transpire. Expanding the Dagger SDK to be used in multiple programs is very important, in order to encourage the integration and adoption, not only of Dagger itself, but also of gesture recognition technology on the whole.

9.2 Attribute Expansion

The Dagger SDK was also designed with the intention of being fairly easy to modify and expand. It is relatively straight-forward to add additional attributes to be used in the definition and recognition of patterns. As such, the expansion of the attribute set would be useful in order to allow developers an even greater number of options for gestures. However, we must be careful not to over-expand this set, to the point where the development of additional device layers becomes extensively tedious.

9.3 Device Compatibility

One of the primary areas which should be extended upon is the compatibility of the Dagger SDK with multiple devices. Dagger has been designed from the beginning to be used with any device from which the necessary information (the position of tracked objects) can be extracted. However, at present, there is only a provided layer for the Leap Motion device.

Given the growth in camera-based gesture recognition, it is important to expand this to include other devices. The Microsoft Kinect, as the second choice for a development device, is a natural candidate. Its upcoming successor, the Kinect One, is another. As other devices emerge, the ability to adapt them for use with Dagger is important and useful.

9.4 Language Porting

The Dagger SDK is currently provided in the Java language. While this is a popular choice, it is not the only option. Additionally, there are some device SDKs which do not support Java - the Microsoft Kinect's official SDK is available only in C++, C\#, and Visual Basic (though third-party SDKs for the Kinect have been implemented in Java). In order to expand the usefulness of Dagger, porting the material to an additional language would be great step.

Chapter 10: Conclusion

In this thesis, we describe the production of an SDK (Dagger) to be used in gesture recognition. The primary goals - all of which were met - were simplicity, gesture support, and device abstraction. The Dagger SDK supports multiple different types of gestures, including single-trackable gestures, relational gestures, multi-stage gestures, and continuous gestures. There is no direct dependence on a particular device, and Dagger is designed to be expanded to include support for multiple camera-based gesture-recognition devices.

BIBLIOGRAPHY

1. Code Laboratories. CL NUI Platform - kinect preview. dec 2010. http://codelaboratories.com/nui/.
2. Y. Cui and D. Stricker. 3D shape scanning with a kinect. In *ACM SIGGRAPH 2011 Posters*, SIGGRAPH '11, pages 57:1-57:1, New York, NY, USA, 2011. ACM.
3. D. C. Engelbart. XY position indicator for a display system, Nov. 17 1970. US Patent 3,541,541.
4. L. Gallo, A. Placitelli, and M. Ciampi. Controller-free exploration of medical image data: Experiencing the kinect. In *Computer-Based Medical Systems (CBMS), 2011 24th International Symposium on*, pages 1-6, june 2011.
5. G. E. Gerpheide. Methods and apparatus for data input, Apr. 19 1994. US Patent 5,305,017.
6. R. Held, A. Gupta, B. Curless, and M. Agrawala. 3D puppetry: a kinect-based interface for 3D animation. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, UIST '12, pages 423-434, New York, NY, USA, 2012. ACM.
7. A. Jaimes and N. Sebe. Multimodal human-computer interaction: A survey.*Comput. Vis. Image Underst.*, 108(1-2):116-134, Oct. 2007.
8. E. Johnson. Touch displays: A programmed man-machine interface. *Ergonomics*, 10(2):271-277, 1967.
9. W. Kadous et al. Grasp: Recognition of australian sign language using instrumented gloves. *Unpublished manuscript*, University of New South Wales, Sydney, Australia. Retrieved October, 1:2002, 1995.
10. Leap Motion, Inc. Leap motion developer. sep 2013. https://developer.leapmotion.com/.
11. J. Lee. Hacking the nintendo wii remote. *Pervasive Computing*, IEEE, 7(3):39-45, july-sept. 2008.
12. Microsoft Corporation. Kinect for Windows. sep 2013. http://www.microsoft.com/en-us/kinectforwindows/.
13. B. Moriarty, E. Lennon, F. DiCola, K. Buzby, M. Manzella, and E. Hromada. Utilizing depth based sensors and customizable software frameworks for experiential application. *Procedia Computer Science*, 12(0):200-205, 2012. Complex Adaptive Systems 2012.
14. B. A. Myers. A brief history of human-computer interaction technology. *interactions*, 5(2):44-54, Mar. 1998.
15. R. E. Nippoldt. Trackball mechanism, Aug. 28 1990. US Patent 4,952,919.
16. OpenKinect. OpenKinect. mar 2012. http://openkinect.org/.
17. OpenNI. OpenNI. 2013. http://www.openni.org.

18. Z. Ren, J. Meng, J. Yuan, and Z. Zhang. Robust hand gesture recognition with kinect sensor. In *Proceedings of the 19th ACM international conference on Multimedia*, MM '11, pages 759-760, New York, NY, USA, 2011. ACM.
19. K. Sung. Recent videogame console technologies. *Computer*, 44(2):91-93, feb. 2011.
20. M. Turk. Gesture recognition. *Handbook of Virtual Environment Technology*, 2001.
21. T. Yamanami, T. Funahashi, and T. Senda. Position detecting apparatus, Nov. 7 1989. US Patent 4,878,553.
22. T. G. Zimmerman. Optical ex sensor, Sept. 17 1985. US Patent 4,542,291.
23. T. G. Zimmerman, J. Lanier, C. Blanchard, S. Bryson, and Y. Harvill. A hand gesture interface device. *SIGCHI Bull.*, 17(SI):189-192, May 1986.

Appendix A: Class and UML Diagrams

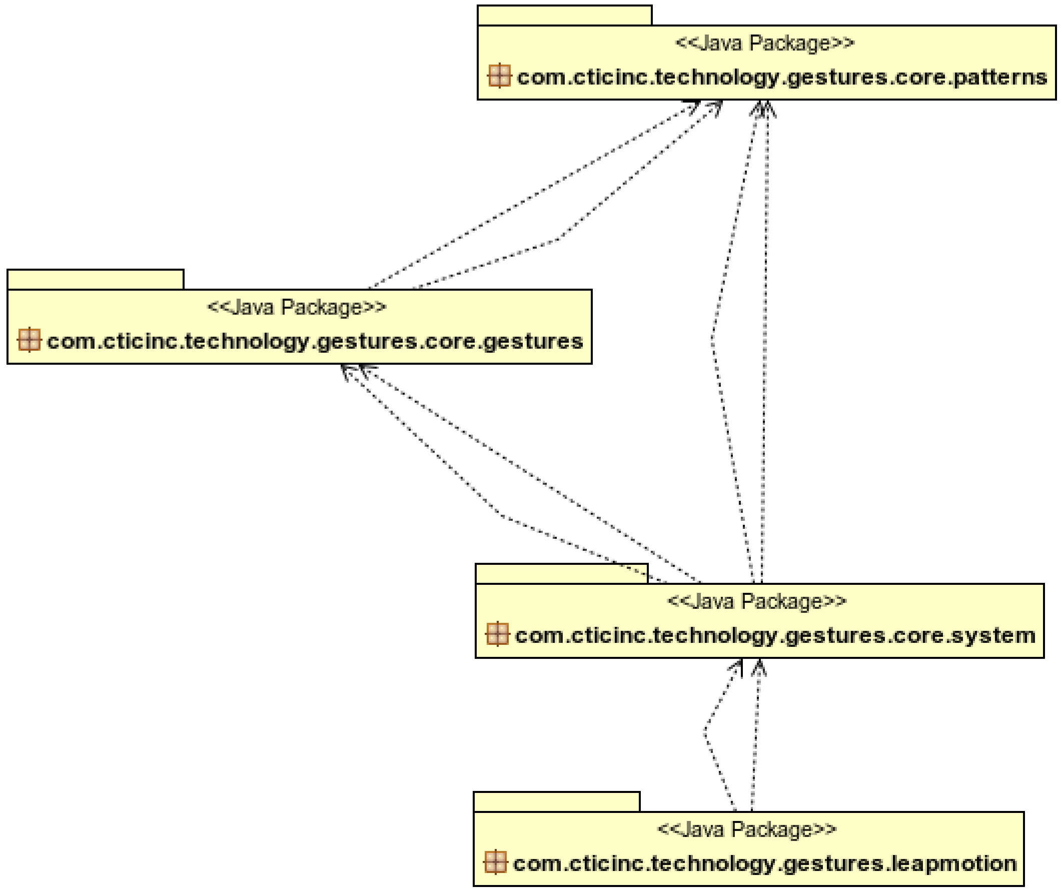

<<Java Package>>
com.cticinc.technology.gestures.core.patterns
<<Java Package>>
com.cticinc.technology.gestures.core.gestures
<<Java Package>>
com.cticinc.technology.gestures.core.system
<<Java Package>>
com.cticinc.technology.gestures.leapmotion

Figure A.1: Full SDK UML

The UML and class diagram for the SDK at a package level. Note: the "common" package has been omitted, since it is, by design, included in every package.

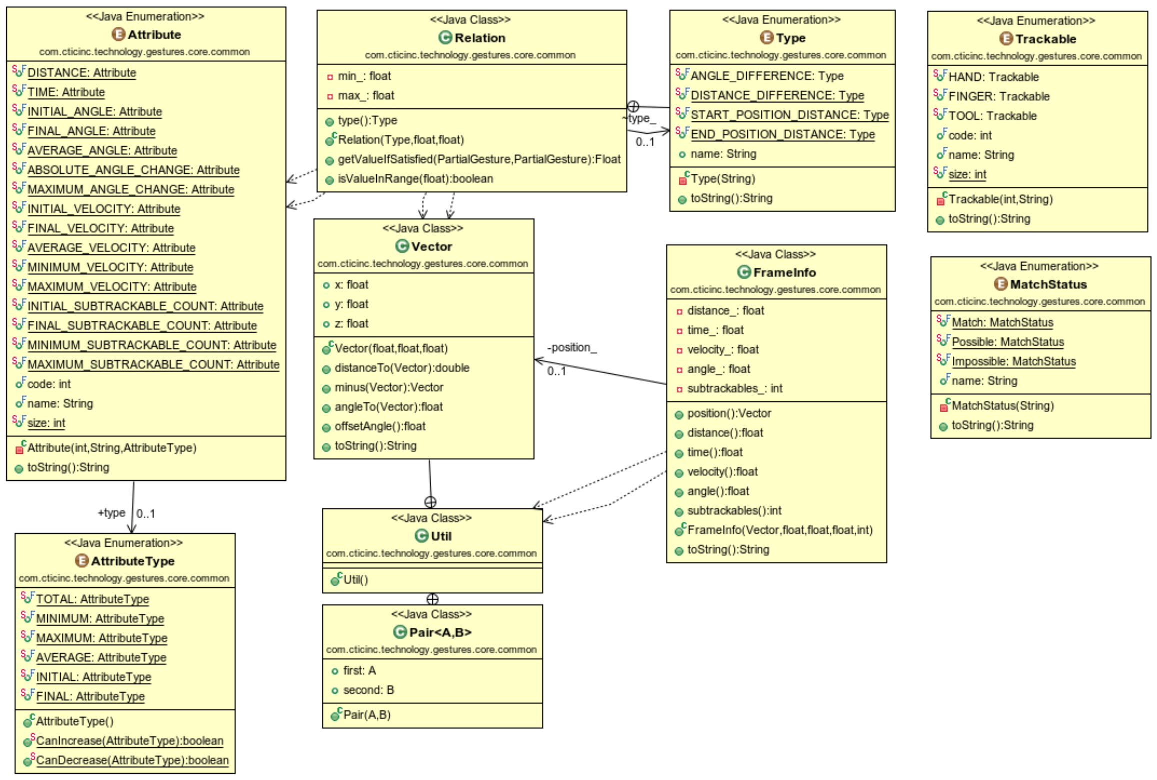
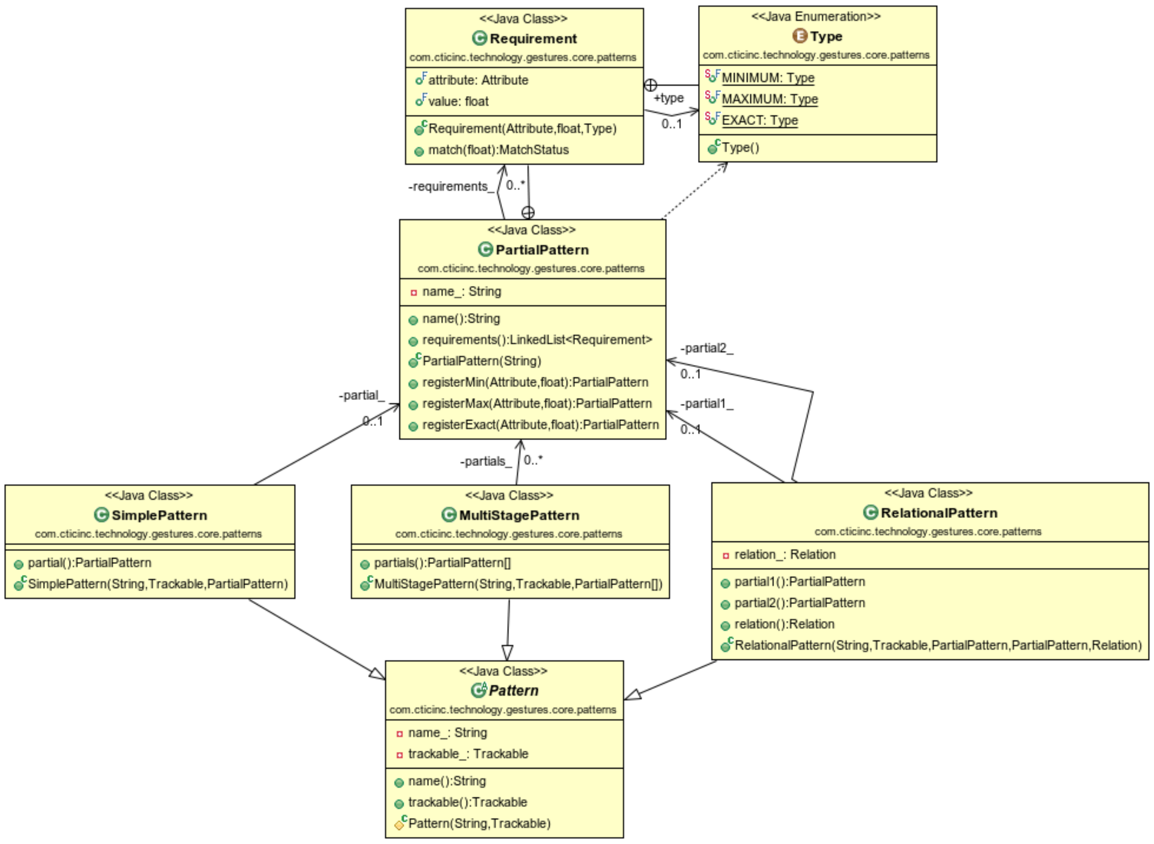


Figure A.2: Common Package UML

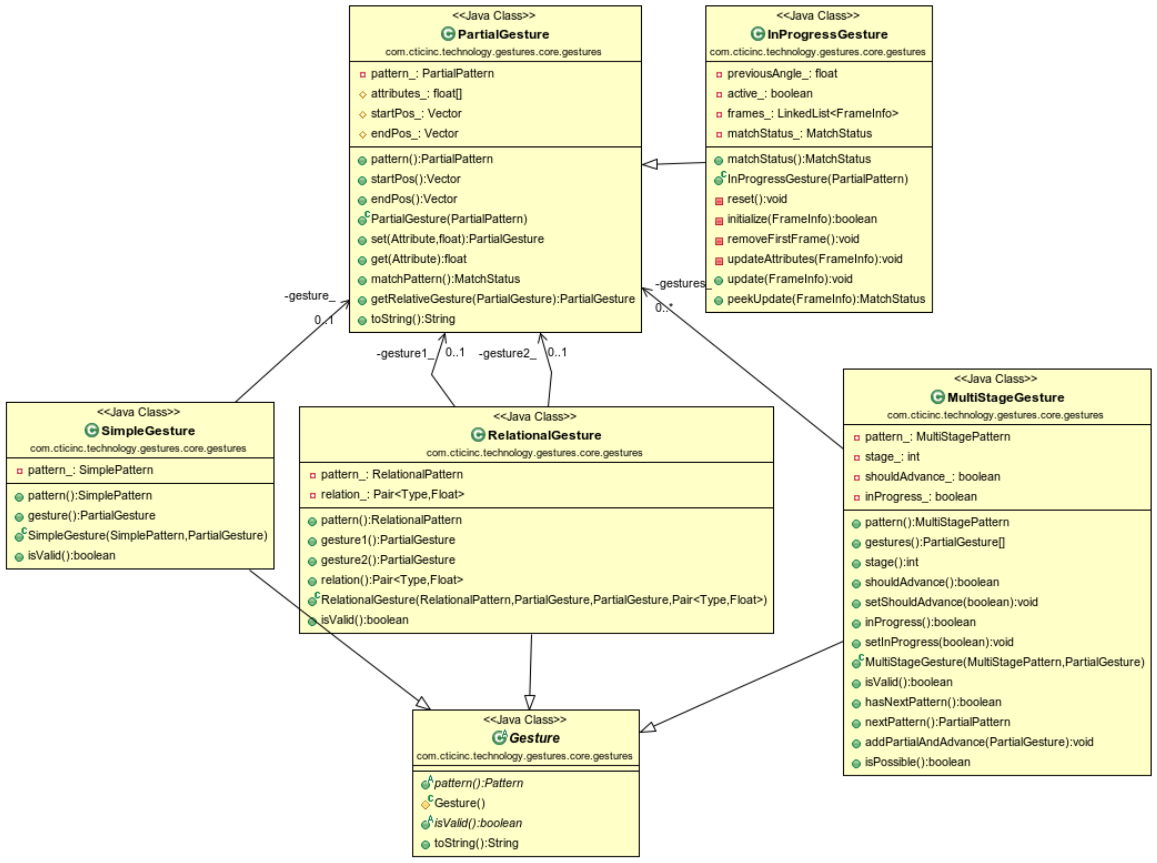
The common package UML. Includes classes which are used in all other packages, and are frequently enums or simple concepts.



<<Java Class>>
Requirement
com.cticinc.technology.gestures.core.patterns
attribute: Attribute
value: float
Requirement(Attribute,float,Type)
match(float):MatchStatus
<<Java Enumeration>>
Type
com.cticinc.technology.gestures.core.patterns
MINIMUM: Type
MAXIMUM: Type
EXACT: Type
Type()
+type
0..1
-requirements_
0..*
<<Java Class>>
PartialPattern
com.cticinc.technology.gestures.core.patterns
name_: String
name():String
requirements():LinkedList<Requirement>
PartialPattern(String)
registerMin(Attribute,float):PartialPattern
registerMax(Attribute,float):PartialPattern
registerExact(Attribute,float):PartialPattern
-partial2_
0..1
-partial1_
0..1
-partial_
0..1
-partials_
0..*
<<Java Class>>
SimplePattern
com.cticinc.technology.gestures.core.patterns
partial():PartialPattern
SimplePattern(String,Trackable,PartialPattern)
<<Java Class>>
MultiStagePattern
com.cticinc.technology.gestures.core.patterns
partials():PartialPattern[]
MultiStagePattern(String,Trackable,PartialPattern[])
<<Java Class>>
RelationalPattern
com.cticinc.technology.gestures.core.patterns
relation_: Relation
partial1():PartialPattern
partial2():PartialPattern
relation():Relation
RelationalPattern(String,Trackable,PartialPattern,PartialPattern,Relation)
<<Java Class>>
Pattern
com.cticinc.technology.gestures.core.patterns
name_: String
trackable_: Trackable
name():String
trackable():Trackable
Pattern(String,Trackable)

Figure A.3: Patterns Package UML

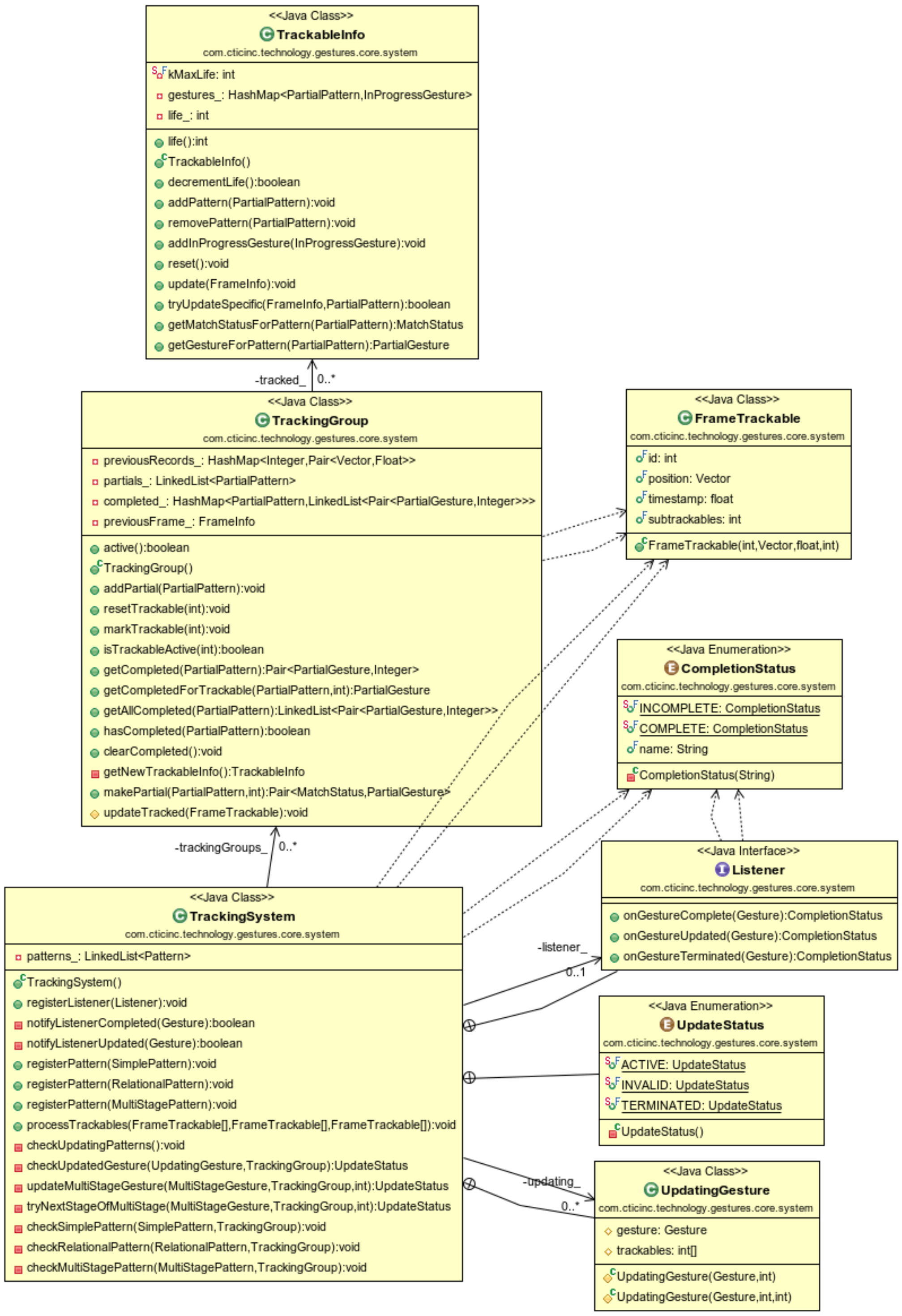
The patterns package UML. Includes all classes relating to Patterns and PartialPatterns, and depends only upon the common package.



<<Java Class>>
PartialGesture
com.cticinc.technology.gestures.core.gestures
pattern_: PartialPattern
attributes_: float[]
startPos_: Vector
endPos_: Vector
pattern():PartialPattern
startPos():Vector
endPos():Vector
PartialGesture(PartialPattern)
set(Attribute,float):PartialGesture
get(Attribute):float
matchPattern():MatchStatus
getRelativeGesture(PartialGesture):PartialGesture
toString():String
<<Java Class>>
InProgressGesture
com.cticinc.technology.gestures.core.gestures
previousAngle_: float
active_: boolean
frames_: LinkedList<FrameInfo>
matchStatus_: MatchStatus
matchStatus():MatchStatus
InProgressGesture(PartialPattern)
reset():void
initialize(FrameInfo):boolean
removeFirstFrame():void
updateAttributes(FrameInfo):void
update(FrameInfo):void
peekUpdate(FrameInfo):MatchStatus
-gesture_
0..1
-gesture1_
0..1
-gesture2_
0..1
-gestures_
0..*
<<Java Class>>
SimpleGesture
com.cticinc.technology.gestures.core.gestures
pattern_: SimplePattern
pattern():SimplePattern
gesture():PartialGesture
SimpleGesture(SimplePattern,PartialGesture)
isValid():boolean
<<Java Class>>
RelationalGesture
com.cticinc.technology.gestures.core.gestures
pattern_: RelationalPattern
relation_: Pair<Type,Float>
pattern():RelationalPattern
gesture1():PartialGesture
gesture2():PartialGesture
relation():Pair<Type,Float>
RelationalGesture(RelationalPattern,PartialGesture,PartialGesture,Pair<Type,Float>)
isValid():boolean
<<Java Class>>
MultiStageGesture
com.cticinc.technology.gestures.core.gestures
pattern_: MultiStagePattern
stage_: int
shouldAdvance_: boolean
inProgress_: boolean
pattern():MultiStagePattern
gestures():PartialGesture[]
stage():int
shouldAdvance():boolean
setShouldAdvance(boolean):void
inProgress():boolean
setInProgress(boolean):void
MultiStageGesture(MultiStagePattern,PartialGesture)
isValid():boolean
hasNextPattern():boolean
nextPattern():PartialPattern
addPartialAndAdvance(PartialGesture):void
isPossible():boolean
<<Java Class>>
Gesture
com.cticinc.technology.gestures.core.gestures
pattern():Pattern
Gesture()
isValid():boolean
toString():String

Figure A.4: Gestures Package UML

The gestures package UML. Includes all classes relating to Gestures, PartialGestures, and InProgressGestures, and depends upon the common package and the patterns package.



<<Java Class>>
TrackableInfo
com.cticinc.technology.gestures.core.system
kMaxLife: int
gestures_: HashMap<PartialPattern,InProgressGesture>
life_: int
life():int
TrackableInfo()
decrementLife():boolean
addPattern(PartialPattern):void
removePattern(PartialPattern):void
addInProgressGesture(InProgressGesture):void
reset():void
update(FrameInfo):void
tryUpdateSpecific(FrameInfo,PartialPattern):boolean
getMatchStatusForPattern(PartialPattern):MatchStatus
getGestureForPattern(PartialPattern):PartialGesture
-tracked_
0..*
<<Java Class>>
TrackingGroup
com.cticinc.technology.gestures.core.system
previousRecords_: HashMap<Integer,Pair<Vector,Float>>
partials_: LinkedList<PartialPattern>
completed_: HashMap<PartialPattern,LinkedList<Pair<PartialGesture,Integer>>>
previousFrame_: FrameInfo
active():boolean
TrackingGroup()
addPartial(PartialPattern):void
resetTrackable(int):void
markTrackable(int):void
isTrackableActive(int):boolean
getCompleted(PartialPattern):Pair<PartialGesture,Integer>
getCompletedForTrackable(PartialPattern,int):PartialGesture
getAllCompleted(PartialPattern):LinkedList<Pair<PartialGesture,Integer>>
hasCompleted(PartialPattern):boolean
clearCompleted():void
getNewTrackableInfo():TrackableInfo
makePartial(PartialPattern,int):Pair<MatchStatus,PartialGesture>
updateTracked(FrameTrackable):void
<<Java Class>>
FrameTrackable
com.cticinc.technology.gestures.core.system
id: int
position: Vector
timestamp: float
subtrackables: int
FrameTrackable(int,Vector,float,int)
<<Java Enumeration>>
CompletionStatus
com.cticinc.technology.gestures.core.system
INCOMPLETE: CompletionStatus
COMPLETE: CompletionStatus
name: String
CompletionStatus(String)
-trackingGroups_
0..*
<<Java Interface>>
Listener
com.cticinc.technology.gestures.core.system
onGestureComplete(Gesture):CompletionStatus
onGestureUpdated(Gesture):CompletionStatus
onGestureTerminated(Gesture):CompletionStatus
<<Java Class>>
TrackingSystem
com.cticinc.technology.gestures.core.system
patterns_: LinkedList<Pattern>
TrackingSystem()
registerListener(Listener):void
notifyListenerCompleted(Gesture):boolean
notifyListenerUpdated(Gesture):boolean
registerPattern(SimplePattern):void
registerPattern(RelationalPattern):void
registerPattern(MultiStagePattern):void
processTrackables(FrameTrackable[],FrameTrackable[],FrameTrackable[]):void
checkUpdatingPatterns():void
checkUpdatedGesture(UpdatingGesture,TrackingGroup):UpdateStatus
updateMultiStageGesture(MultiStageGesture,TrackingGroup,int):UpdateStatus
tryNextStageOfMultiStage(MultiStageGesture,TrackingGroup,int):UpdateStatus
checkSimplePattern(SimplePattern,TrackingGroup):void
checkRelationalPattern(RelationalPattern,TrackingGroup):void
checkMultiStagePattern(MultiStagePattern,TrackingGroup):void
-listener_
0..1
<<Java Enumeration>>
UpdateStatus
com.cticinc.technology.gestures.core.system
ACTIVE: UpdateStatus
INVALID: UpdateStatus
TERMINATED: UpdateStatus
UpdateStatus()
-updating_
0..*
<<Java Class>>
UpdatingGesture
com.cticinc.technology.gestures.core.system
gesture: Gesture
trackables: int[]
UpdatingGesture(Gesture,int)
UpdatingGesture(Gesture,int,int)

Figure A.5: System Package UML

The system package UML. Includes all classes related to the system core and the process of matching patterns. Depends on all packages, except the Leap Motion package.

Appendix B: Dagger SDK for NASA Worldwind

```
private class Listener implements TrackingSystem.Listener {
  private TrackingSystem trackingSystem_;
  // Patterns
  private SimplePattern panPattern_;
  private RelationalPattern zoomPattern_;
  private MultiStagePattern cursorGrabAndDrop_;

  private double previousZoomDistance_;
  private Util.Vector cursorPos_;

  protected TrackingSystem trackingSystem() {
      return trackingSystem_;
  }

  Listener() {
    super();

    trackingSystem_ = new TrackingSystem();
    initPatterns();
    registerPatterns();

    trackingSystem_.registerListener(this);
    cursorPos_ = null;
  }

  private void initPatterns() {
    PartialPattern swiftPan =
        new PartialPattern("Swift Pan Partial")
        .registerMin(Attribute.DISTANCE, 125.f)
        .registerMax(Attribute.MAXIMUM_ANGLE_CHANGE, 40.f);
    panPattern_ =
        new SimplePattern("Swift Pan", Trackable.HAND, swiftPan);

    PartialPattern zoom = new PartialPattern("Zoom Partial")
        .registerMin(Attribute.DISTANCE, 20.f)
        .registerMax(Attribute.MAXIMUM_ANGLE_CHANGE, 20.f);
    Relation zoomRelation = new Relation(
        Relation.Type.ANGLE_DIFFERENCE, 160.f, 200.f);
    zoomPattern_ = new RelationalPattern(
        "Zoom", Trackable.FINGER, zoom, zoom, zoomRelation);

    PartialPattern toggleCursor =
        new PartialPattern("Toggle Cursor Partial")
        .registerMin(Attribute.INITIAL_SUBTRACKABLE_COUNT, 5)
        .registerMin(Attribute.FINAL_SUBTRACKABLE_COUNT, 5)
        .registerMax(Attribute.DISTANCE, 100.f)
        .registerMin(Attribute.TIME, 1500000);
    PartialPattern moveCursor =
        new PartialPattern("Move Cursor Partial")
        .registerMin(Attribute.INITIAL_SUBTRACKABLE_COUNT, 4)
        .registerMin(Attribute.FINAL_SUBTRACKABLE_COUNT, 4);
    PartialPattern padding =
        new PartialPattern("Padding Partial")
        .registerMax(Attribute.INITIAL_SUBTRACKABLE_COUNT, 3)
        .registerMin(Attribute.FINAL_SUBTRACKABLE_COUNT, 2)
        .registerMax(Attribute.TIME, 250000);
    PartialPattern grabbedMove =
        new PartialPattern("Grabbed Move Partial")
        .registerMax(Attribute.INITIAL_SUBTRACKABLE_COUNT, 1)
        .registerMax(Attribute.FINAL_SUBTRACKABLE_COUNT, 1);
    cursorGrabAndDrop_ = new MultiStagePattern(
        "Cursor Grab and Drop",
        Trackable.HAND,
        new PartialPattern[] {
            toggleCursor, moveCursor, padding, grabbedMove });
  }

  private void registerPatterns() {
    trackingSystem_.registerPattern(panPattern_);
    trackingSystem_.registerPattern(zoomPattern_);
    trackingSystem_.registerPattern(cursorGrabAndDrop_);
  }

  private CompletionStatus onPanGesture(SimpleGesture simple,
                                        boolean isUpdate) {
    assert(!isUpdate);
    controller_.panMap(
        simple.gesture().get(Attribute.AVERAGE_VELOCITY),
        simple.gesture().get(Attribute.AVERAGE_ANGLE));
    return CompletionStatus.COMPLETE;
  }

  private CompletionStatus onZoomGesture(
      RelationalGesture relational, boolean isUpdate) {
    double distance = 0.;
    double startDistance =
        Math.abs(relational.gesture1().startPos().distanceTo(
            relational.gesture2().startPos()));
    double endDistance =
        Math.abs(relational.gesture1().endPos().distanceTo(
            relational.gesture2().endPos()));
    if (!isUpdate) {
      previousZoomDistance_ = endDistance - startDistance;
      distance = previousZoomDistance_ - 20.f;
    } else {
      distance =
          endDistance - startDistance - previousZoomDistance_;
      previousZoomDistance_ = endDistance - startDistance;
    }

    controller_.zoomMap(distance / 10. * -1.);
    return CompletionStatus.INCOMPLETE;
  }

  private CompletionStatus onCursorGrabAndDropGesture(
      MultiStageGesture gesture, boolean isUpdate) {
    if (gesture.stage() == 0) {  // Toggle
      assert(!isUpdate);
      cursorPos_ = gesture.gestures()[0].endPos();
      controller_.showCursor();
      // Enter move cursor stage
      return CompletionStatus.COMPLETE;
    } else if (gesture.stage() == 1) {  // Move cursor stage
      Util.Vector endPos = gesture.gestures()[1].endPos();
      controller_.moveCursor((endPos.x - cursorPos_.x) / 10.f,
                             (endPos.y - cursorPos_.y) / 10.f);
      cursorPos_ = endPos;
      return CompletionStatus.INCOMPLETE;
    } else if (gesture.stage() == 2) {  // padding
      return CompletionStatus.INCOMPLETE;
    } else if (gesture.stage() == 3) {  // grab
      if (!isUpdate)
        controller_.pickUp();
      Util.Vector endPos = gesture.gestures()[3].endPos();
      controller_.moveCursor((endPos.x - cursorPos_.x) / 10.f,
                             (endPos.y - cursorPos_.y) / 10.f);
      cursorPos_ = endPos;
      return CompletionStatus.INCOMPLETE;
    }
    assert(false);
    return null;
  }

  @Override
  public CompletionStatus onGestureComplete(Gesture gesture) {
    if (gesture.pattern() == swiftPanPattern_) {
      return onSwiftPanGesture((SimpleGesture)gesture, false);
    } else if (gesture.pattern() == slowPanPattern_) {
      return onSlowPanGesture((SimpleGesture)gesture, false);
    } else if (gesture.pattern() == zoomPattern_) {
      return onZoomGesture((RelationalGesture)gesture, false);
    } else if (gesture.pattern() == cursorGrabAndDrop_) {
      return onCursorGrabAndDropGesture(
          (MultiStageGesture)gesture, false);
    }
    assert(false);
    return CompletionStatus.COMPLETE;
  }

  @Override
  public CompletionStatus onGestureUpdated(Gesture gesture) {
    if (gesture.pattern() == swiftPanPattern_) {
      return onSwiftPanGesture((SimpleGesture)gesture, true);
    } else if (gesture.pattern() == slowPanPattern_) {
      return onSlowPanGesture((SimpleGesture)gesture, true);
    } else if (gesture.pattern() == zoomPattern_) {
      return onZoomGesture((RelationalGesture)gesture, true);
    } else if (gesture.pattern() == cursorGrabAndDrop_) {
      return onCursorGrabAndDropGesture(
          (MultiStageGesture)gesture, true);
    }
    assert(false);
    return CompletionStatus.COMPLETE;
  }

  @Override
  public CompletionStatus onGestureTerminated(Gesture gesture) {
    if (gesture.pattern() == cursorGrabAndDrop_) {
      MultiStageGesture msg = (MultiStageGesture) gesture;
      if (msg.stage() == 1) {
        controller_.hideCursor();
      } else if (msg.stage() == 2) {
        controller_.hideCursor();
      } else if (msg.stage() == 3) {
        controller_.hideCursor();
      }
      cursorPos_ = null;
    }
    return null;
  }
}
```