Sketch IPT Common Sketch Module

Sketch Recognition Lab @ Texas A&M University
Deep Green Project Team

November 10, 2008

©Texas A&M University, 2008-2011

Revision History		
Date	Author	Version / Change
2008, August 12	Joshua Johnston	1.0 - Original
2008, September 08	Joshua Johnston	2.0 - Statement of Work, Aliases, Shape Attributes, N-Best list
2008, September 26	Joshua Johnston	3.0 - Clarification of axes system for `x` and `y` for Point, Sequence diagram for add stroke/recognize
2008, October 6	Joshua Johnston	4.0 - Change `NBestList` to `RecognitionResult`, update sequence diagram, remove references to XML commands
2008, November 10	Joshua Johnston	5.0 - Clarification on `x` and `y` semantics.

Contents

1 Introduction

This document outlines the statement of work, functional description, and interface specifications for the Sketch IPT Common Sketch Module developed by the Sketch Recognition Lab @ Texas A&M University. Questions, comments, and suggestions should be forwarded to:

Sketch Recognition Lab @ Texas A&M University
Deep Green Project Team
deepGreen@cs.tamu.edu
(979) 845-7143 (Sketch Lab phone)

2 Statement of Work

The SOW for Sketch Recognition Lab @ Texas A&M University is defined in `SketchRecognitionCommonSOWv3.doc` as distributed on Fri, Sep 5, 2008 at 12:02 PM.

3 Functional Description

Texas A&M's contribution to the Deep Green project is a core system that performs sketch recognition. Our system takes, as input, a set of strokes. These strokes are made of points of raw digital ink, expressed in their simplest form as ⟨`X, Y, TIMESTAMP`⟩ tuples. Our sketch algorithms work on these raw strokes at multiple levels.

Vision algorithms, like those provided by Dr. Christine Alvarado's team at Harvey Mudd College, look for constructions of raw ink points in relation to one another. If these relations have certain characteristics, symbol recognition can be performed (this set of ink points "looks like" this symbol).

Segmentation/corner finding algorithms examine the raw ink and divide the raw strokes into primitives. Examples of primitives include line segments, circles, ellipses, arcs and curves, helixes, and spirals. A primitive is a simple shape that cannot be composed well of other shapes. These primitives are the building blocks for more complex constructs.

Hierarchical shape recognizers build up complex shapes using primitives and other shapes. Shape subparts are put together in specific ways defined by the domain.

3.1 Sketch Recognition Process

Figure 1 gives an illustration of the sketch recognition process. User input consists of raw strokes, which are series of points with ⟨`X, Y, TIMESTAMP`⟩ information. Additional information might include the pressure or tilt of the pen. After various pre-processing techniques to clean up the strokes, they are fed into the sketch recognition process. Low-level recognizers break strokes into primitives, such as lines and arcs, using corner finding techniques and primitive identification. Higher-level recognition combines low-level shape recognition results (or possibly raw strokes, in the case of vision algorithms) in an iterative, hierarchical manner. This figure is taken Hammond, et al., 2008.[1]

Only the sketch module should handle stroke pre-processing. No modification should be made to the stroke data before it is sent to the sketch module.

[1] Hammond, Tracy, Brian Eoff, Brandon Paulson, Aaron Wolin, Katie Dahmen, Joshua Johnston, and Pankaj Rajan. Free-Sketch Recognition: Putting the CHI in Sketching. In *Proceedings of the 2008 Conference on Human Factors in Computing Systems* (CHI), 2008

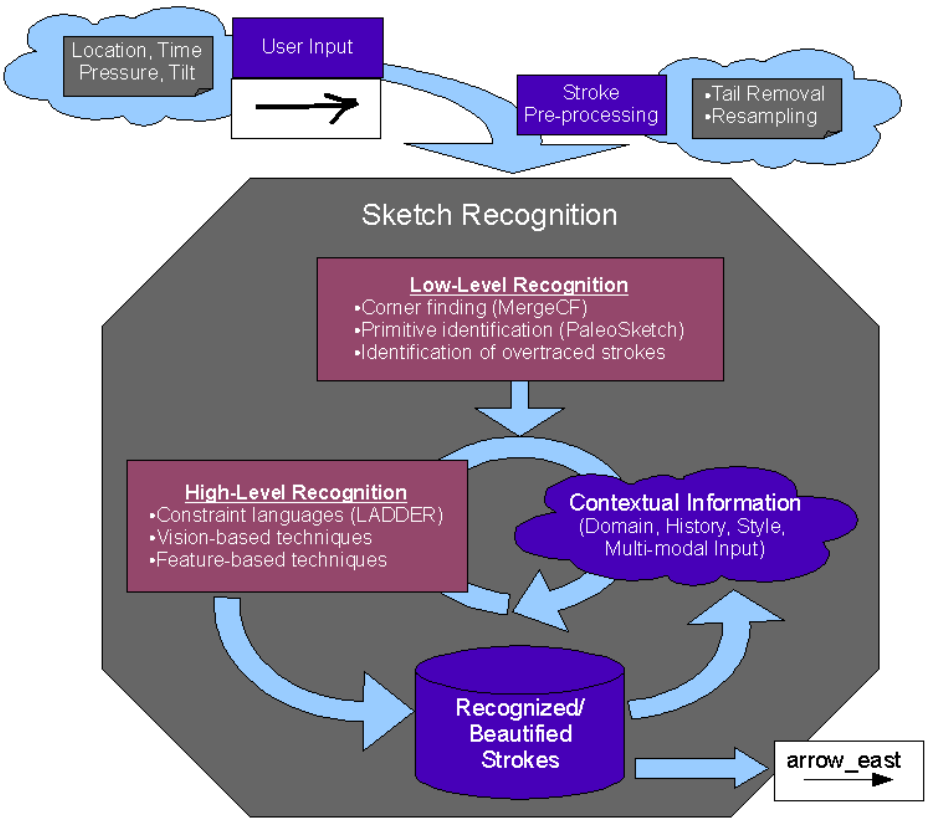

Figure 1: Overview of sketch recognition process, which is very cyclical and hierarchical in nature. The illustrative input here might contain three strokes, forming an arrow. The single output is a beautified and recognized arrow, with semantic identification attached.

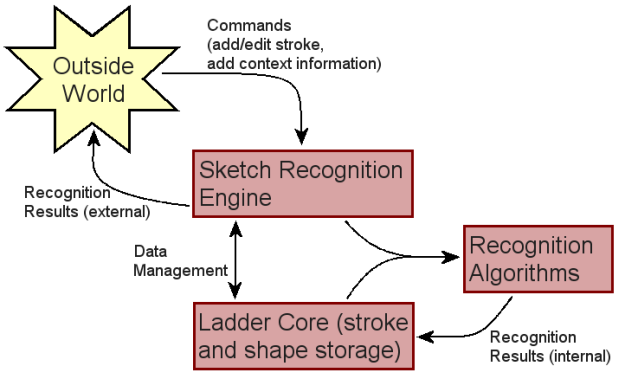

Figure 2: Flow of information, at a high level, in to, out of, and within the sketch recognition module.

3.2 Functional Flow

Figure 2 illustrates the input to and output from the sketch recognition module, including management of stroke information and tasks such as drawing and editing strokes, providing contextual information, and receiving recognition results. To the sketch recognition engine, the outside world is a black box. The outside world (the user interface that interacts with the sketcher, collects stroke data, and displays recognized shapes and symbols) communicates with the engine by issuing simple commands. These commands allow the user interface to add strokes to a sketch, request the sketch be recognized, modify/edit the sketch, and provide contextual information to help the sketch recognition engine perform more accurate recognition.

3.3 Example Recognition Scenario

Figure 3 provides an example scenario of sketch recognition, including the movement of raw stroke data through the sketch recognition process and how it is turned into a fully recognized symbol. The recognition process takes raw ink strokes. A stroke is a series of points consisting of $\langle$`X, Y, TIMESTAMP`$\rangle$ information (possibly with more attributes, such as pen pressure or tilt). It might be the case that the strokes (one for the rectangle, two for the $\times$, and one for the airborne modifier) are passed in one at a time, or in bulk all at once.

The raw strokes are segmented into primitives using corner finding techniques, stroke segmentation, and primitive recognition. This is a fairly simple sketch, consisting of mainly line segment primitives (4 line segments in the rectangle, even though it was drawn with one raw stroke, and 2 line segments for the $\times$). There are also a few curves in the airborne modifier. Additional information that could have also been sketched are the echelon and unit labels, though we leave these out for simplicity's sake.

The primitives are combined together into a set of initial, low-level shapes. Though we make a distinction here between low- and high-level shapes, within the sketch recognition process there is none. Everything above raw strokes is just a shape, regardless of its complexity. For clarity, we only include the low-level shapes as they would be represented if stroke grouping and shape recognition were perfect. Of course things aren't this simple and don't work this well, but there's no need to list all the possible incorrect alternatives or near-misses. We end up with one rectangle, one $\times$, and one airborne modifier.

The shape recognition continues in an iterative manner, constructing a hierarchy of shapes from the bottom up. We might start by recognizing the rectangle from four segments, then an $\times$ from two segments, combine the rectangle and $\times$ to form an infantry unit, recognize the curves as the airborne modifier, and then combine the infantry unit with the airborne to form the final airborne infantry shape.

The output of the sketch recognition system is an n-best list identifying the possible shapes that might be present in the provided strokes. It might be the case that the full "friendly airborne infantry" symbol has the most confidence since it uses all the strokes. Tied for second might come just an infantry unit and just

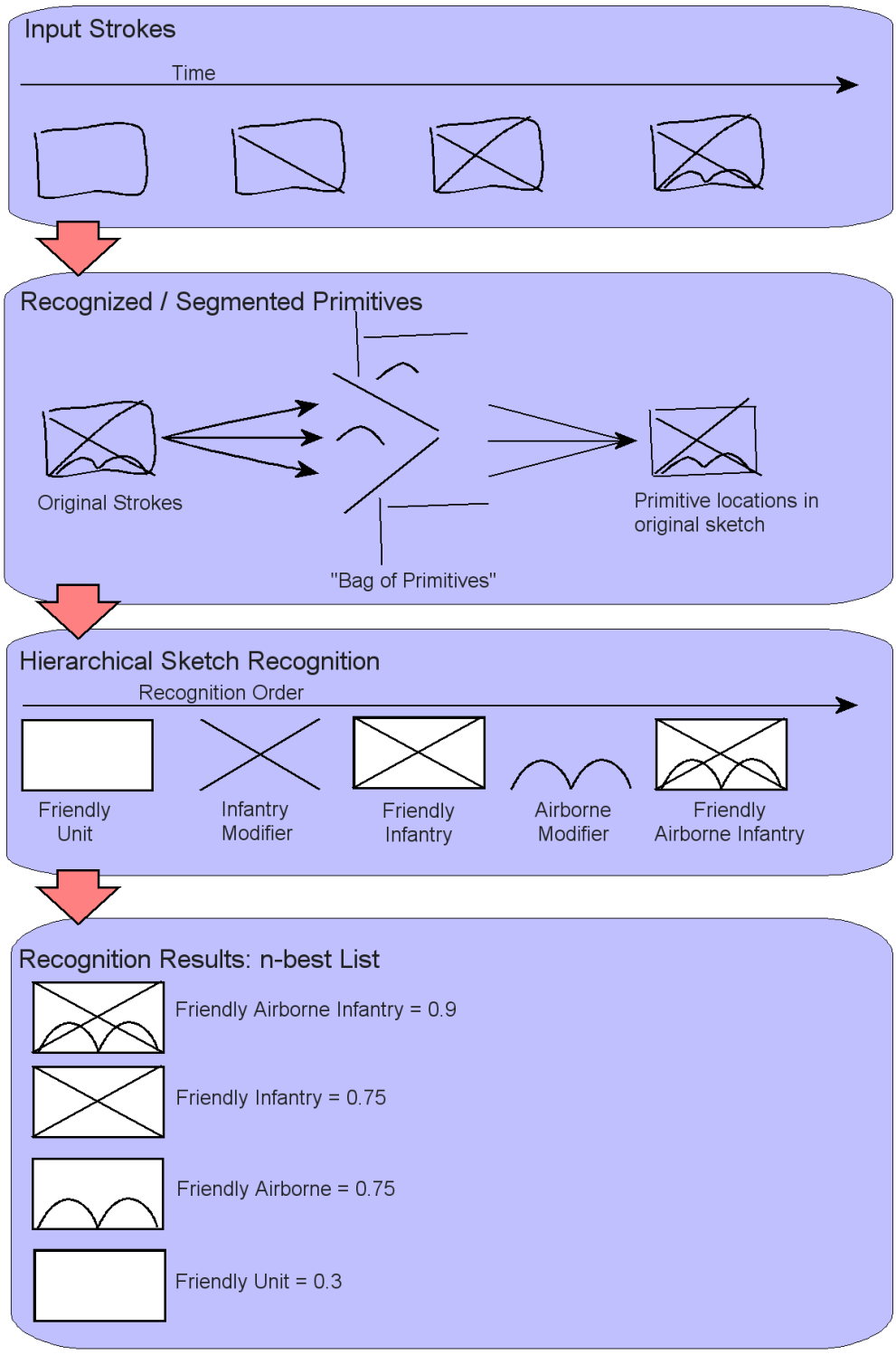

Figure 3: Example sketch recognition scenario. Strokes might be drawn in the indicated order, perfectly segmented into the given primitives, perfectly recognized in the given order, and then returned as shape interpretations in the given n-best list (with made up, hypothetical confidence values).

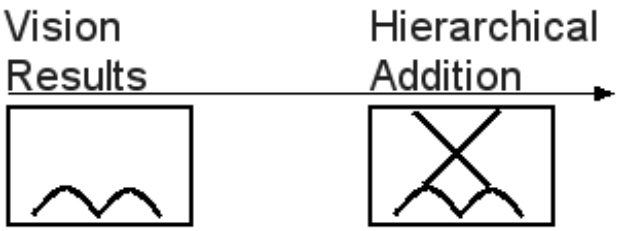

Figure 4: Visual algorithms provide a recognized "airborne unit" with high confidence, saving the hierarchical shape recognition process the time and effort of finding a rectangle and piecing the shape together one sub-shape at a time. From the visual algorithm's recognition result, we might find other parts of the symbol iteratively, like the infantry modifier, and add them hierarchically.

an airborne unit. These are possibilities that have a bit lower confidence since not all the strokes are used. Of course, this is assuming we have an algorithm that will boost the confidence for shapes that use more strokes at a time. Perhaps the infantry modifier and/or airborne modifier are stray strokes, we don't know. At the very lowest level of confidence might be the sole rectangle/friendly unit symbol. We might also just include the airborne modifier, or just the $\times$ as possible shapes in the n-best list. Later in the life-cycle of the project, we might also include an interpretation of a "destroy this friendly airborne unit", with the $\times$ being interpreted as a destroy task modifier rather than an infantry modifier.

3.4 Role of Vision (and other) Algorithms

It also might be the case that a non-geometric recognizer, such as the vision-based algorithms provided by Dr. Alvarado's team, can short-circuit some of the iterations needed to construct the full sketch in the above example. For example, as shown in Figure 4, we might start with the vision algorithm recognizing a friendly airborne unit, with the hierarchical recognition only adding the infantry modifier. This example makes sense because an airborne unit, which is made up of curves, might be more easily recognized by a vision algorithm than a geometric constraint algorithm. It's easier to write geometric constraints for line segments than for curves.

4 Interfaces

The following section lays out a general overview of data representation within the sketch recognition module, as well as interfaces for input and output communication with the sketch module.

4.1 Data Representation

This section describes the internal data representation used within the sketch recognition module. Its purpose is to elucidate further the way the sketch recognition module views data, and the way input strokes are handled and turned into recognized symbols.

4.1.1 Points

A `POINT` is an ⟨`X, Y, TIMESTAMP`⟩ tuple.

- `X` – A double precision floating point value. In our system, this is the number of pixels from the right edge of the window the point was drawn in. However, this value is relative to its neighbors, and the sketch system does not care what the value represents or what its units are as long as values are consistent with each other.

- `Y` – A double precision floating point value. In our system, this is the number of pixels from the top edge of the window the point was drawn in. However, this value is relative to its neighbors, and the sketch system does not care what the value represents or what its units are as long as values are consistent with each other.
- `TIMESTAMP` – A long integer (64 bit signed) that represents the time this point was drawn. In our system (using Java), by default, this represents the number of milliseconds elapsed since midnight, January 1, 1970 UTC. However, this value is relative, and the sketch system does not care what the value represents or what its units are, as long as values are consistent with each other.
- `ID` – The unique identifier for this particular point. In our system, we use the Java implementation of a universally unique identifier (UUID).

`X` and `Y` values must be represented in **GLOBAL COORDINATES**, so that zooming in and out and panning around a map doesn't result in coordinate values that are not scaled with each other.

Although the magnitude and absolute values of `X` and `Y` will not affect recognition results, it must be the case that provided `X` and `Y` values adhere to the same axes used by a computer monitor. Values for `X` must be low on the "left" and increase to the "right." Values for `Y` must be low at the "top" and increase toward the "bottom." This convention must be followed because of the assumption that incoming coordinates are captured from Java GUI components, which use the screen coordinate axes.

The system makes no semantic assumptions as to the meaning of the values of the `X` and `Y` coordinates. Any discontinuities in the coordinate system (for example, the Equator, Prime Meridian, and International Date Line if using latitude/longitude) *must* be handled by the user.

4.1.2 Aliases

An `ALIAS` is a named reference to a particular point of importance/interest within a shape. In Deep Green lingo, an alias is a control/anchor point. Aliases are simply wrappers to a `POINT` and an associated name.

- `POINT` – The `POINT` this `ALIAS` refers to.
- `NAME` – The name of this `ALIAS`

4.1.3 Strokes

A `STROKE` is a finite `LIST` of `POINT` information that is drawn by the user and captured in electronic format or digitized.

- `LIST`⟨`POINT`⟩ – List of `POINT`s that make up this stroke.
- `ID` – The unique identifier for this particular point. In our system, we use the Java implementation of a universally unique identifier (UUID).

4.1.4 Shapes

A `SHAPE` is a set of one or more `STROKE`s that has been recognized by a sketch-recognition algorithm and assigned semantic meaning. A `SHAPE` is an interpretation of `STROKE`s into symbols, control measures, and other information conveyed with the pen.

- SET⟨STROKE⟩ – Set of strokes (references or IDs) that this shape is composed of.
- LABEL – Semantic label assigned to this interpretation. Internally, this value will be simple text (e.g. "Friendly Infantry Battalion") that's been used to define that particular shape in our recognizer.
- CONFIDENCE – A decimal value $\in [0 \dots 1]$ representing the confidence (posterior probability) of the sketch engine in its recognition.
- SET⟨ALIASES⟩ – Shapes will contain a set of aliases, which will store "control points" as defined in MIL STD 2525 b. Aliases can be accessed by name or as a Collection as a whole.
- ATTRIBUTES – Various other attributes that can be set on a SHAPE. In particular, this is where the SIDC/Mole Code will be placed, as well as any symbol modifiers. Attributes exist in a map. They are String attributes with String keys (java.lang.String). To access the different attributes we'll place into the map, static keys are defined in the interface class `edu.tamu.deepGreen.DeepGreenSketchRecognizer.java`. See the API for more information.
- ID – The unique identifier for this particular point. In our system, we use the Java implementation of a universally unique identifier (UUID).

4.2 Input Commands

Communication from the outside world into the sketch recognition module is accomplished by issuing different commands to the module (Figure 2). Commands are issued via method calls into the sketch recognition interface (`DeepGreenSketchRecognizer`).

This specification assumes that the user interface will handle most editing gestures and operations, like moving and deleting strokes.

4.2.1 Adding Strokes

Add the given stroke or list of strokes to the sketch representation in the sketch engine.

ADD_STROKE(STROKE)
ADD_STROKE(LIST⟨STROKE⟩)

4.2.2 Adding Shapes

If a shape is added through speech or other contextual information, you can tell the sketch module so it can use the shape to help aide recognition of other strokes.

ADD_SHAPE(SHAPE)

4.2.3 Deleting Strokes and shapes

Deleting strokes and shapes allows a user or the system to modify the sketch as needed

Currently, editing commands like moving, scaling, or rotating a stroke or shape require deleting the old object and adding the new, modified one. It is assumed at this point in time that editing commands are handled by the user interface.

DELETE_STROKE(STROKE)
DELETE_SHAPE(SHAPE)

4.2.4 Request Recognition

Tell the sketch engine explicitly that you want it to recognize all the collected, unused strokes (though most likely, this has been occurring in the background as strokes are added).

This method will also flag to the sketch engine that you would like the results of recognition when available. This is the only way to tell the sketch engine that you are ready and would like results.

`RECOGNIZE`

4.2.5 Release Interpretations (mark as incorrect)

Flag all the shape interpretations in the list as being incorrectly interpreted by the sketch engine.

Issuing this command will cause the sketch engine to return all strokes used by the specified shape interpretations back into the unused pool, where it can try to classify them again.

This command is most useful when strokes have been classified prematurely, before additional information is available to the sketch recognition engine. This is like an undo for the recognition process.

`RELEASE_INTERPRETATION(LIST⟨SHAPE⟩)`

4.2.6 Accept Interpretations (mark as correct)

Flag all the shape interpretations in the list as being correct and verified.

Issuing this command lets the sketch recognition module know that it should not decompose or reevaluate the indicated interpretations.

`ACCEPT_INTERPRETATION(LIST⟨SHAPE⟩)`

4.3 Output Messages

Output messages are the way that the sketch module communicates information, like recognition results, back to the outside world/user interface. Output can either be synchronized or asynchronous. It's handled all the same within the sketch module, and is solely up to the system interfacing with the sketch module if it wants to wait or not.

4.3.1 Recognition Result

When requesting recognition results, our system will provide a set of multiple interpretations with different levels of confidence. This way, context fusion systems can determine which interpretation is best. We will wrap sketch recognition results in a `RECOGNITIONRESULT` object. These objects store n-best lists of different recognition interpretations, as well as many helper functions for operations on the lists. An n-best list represents all possible recognitions for ONE SET OF STROKES in the sketch, as grouped by the sketch recognition algorithm(s). If recognition is requested on an entire sketch, and/or if the sketch algorithm determines the drawn strokes to fall into disjoint groupings, the system will return a list of `RECOGNITIONRESULT`, one per grouping of strokes.

The list of `RECOGNITIONRESULT` returned from the sketch recognition module will only include items that have changed since the last call to `RECOGNIZE`. All other items in the sketch can be assumed to remain unchanged by any recognition that has occurs. In this manner, a `RECOGNITIONRESULT` itself serves as a change notification.

- `LIST⟨SHAPE⟩` – The list of recognized shapes in the n-best list.
- `ID` - Unique identifier for this `RECOGNITIONRESULT`, using Java's UUID implementation.

4.4 Using the Interfaces

Figure 5 shows a sequence diagram of high-level actions that occur within the sketch recognition engine when making various interface calls.

Strokes are added to the system, one at a time, using the `ADD_STROKE` interface call. These strokes are broken up and segmented by primitive shape recognizers. The primitive shapes are fed, one at a time, into the high level recognition system. This system looks at the definitions of various high level shapes and tries to plug the low-level primitives it's receiving into these "recipes." At the same time, it's trying to determine if primitives that are coming in are to affect shapes that have already been recognized (annotations or more strokes to finish up an incomplete shape) or belong to an entirely new shape. Depending on the results of grouping, new `RecognitionResult` objects (and hence n-best lists) may be created, or recognized shapes might be added to current results.

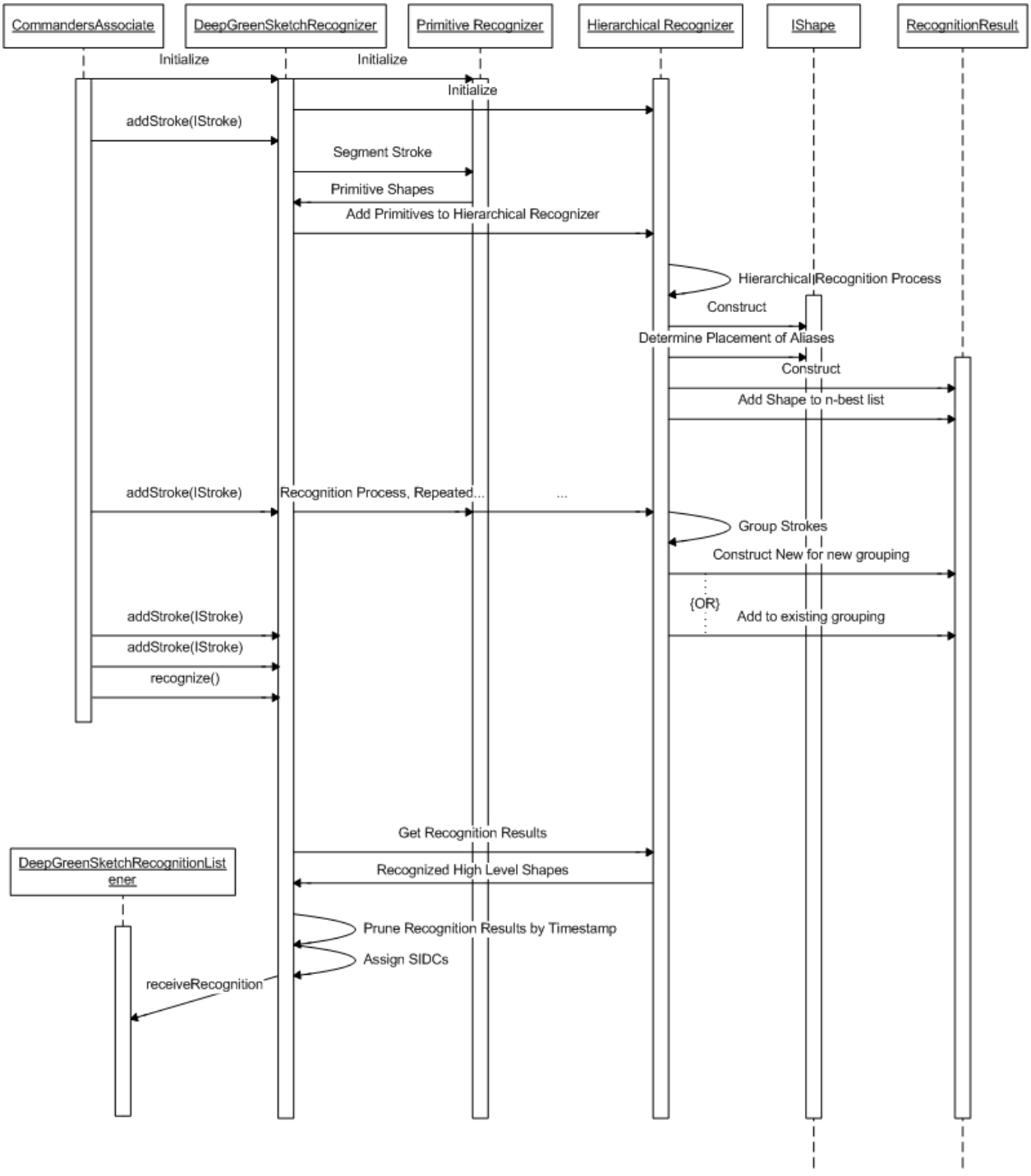

Figure 5: High-level sequence diagram of recognition process as strokes are added to the system, the primitives are processed from those strokes, and recognition results are requested from the system.