4
Drawing the interface

A prototype in UISKEI is composed by **presentation units**, which are user interface containers that can represent a window or a webpage, for example. Elements are then added to the presentation units. An element can be a widget (recognized element) or a scribble (unrecognized element). The elements' creation process by drawing them will be detailed in the next sections, covering all steps from the user drawing a stroke, to the stroke being recognized as a shape and, finally, to the element being created.

Section 4.1 discusses how UISKEI converts the ink stroke to a `Segment` data structure. Section 4.2 describes how the shapes are defined as a string. Section 4.3 presents the `ElementDescriptor` concept to define an element in UISKEI. Section 4.4 details the recognition process. Finally, Section 4.5 enumerates the available elements' properties.

4.1
Segments

When the user starts drawing, he/she may express his/her idea using a single line or several lines. The ink SDK considers that a stroke was created every time the user lifts up the pen, even if he/she is in the middle of a drawing. Multiple strokes can therefore frequently be combined into a single segment. For example, a rectangle can be drawn using a single stroke or more than one, as seen in the figure below (where the dot marks the initial point of each stroke):

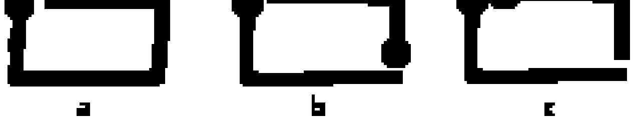

Figure 14: Drawing a rectangle in three different ways.

As can be seen, the leftmost rectangle (Figure 14a) was drawn with a single stroke, while the other ones were drawn with different combinations of two strokes: Figure 14b has the initial point of a stroke near the end point of the other, whilst Figure 14c has the initial point and end point of the strokes near one another. To simplify the shape recognition process, the strokes are converted into a **`Segment`**, which is a list of points with a bounding box. When strokes can be combined, like in Figure 14b and Figure 14c, they are merged into a single `Segment`, only having to change the stroke's direction if needed (as happens in Figure 14c).

Though merging solves part of the problem, the user can still have a drawing with multiple strokes that can't be combined into a single one. For example, if he/she wants to write the word "test" or draw a square with an "X" inside, he/she may end up with three strokes in each drawing that should not be combined, as illustrated in the figure below.

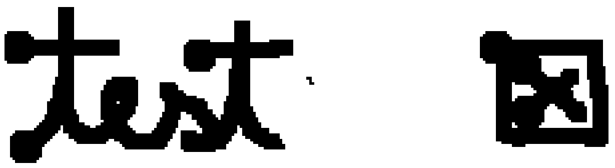

Figure 15: Multiple strokes that should be grouped but not merged.

Besides merging strokes into `Segment`, it is also needed to group strokes that can't be merged. Therefore, a **`MultipleSegments`** is a list of `Segment` with a bounding box.

When the user draws a stroke, it is converted to a `Segment` and added to the current `MultipleSegments` being drawn, with three possible outcomes:

- The new `Segment` is merged with another `Segment` already in the `MultipleSegments` (when its initial point is close enough to the initial/end point of the other segment, like Figure 14b and Figure 14c);
- It is added to the `MultipleSegments` (when it couldn't be merged to another `Segment`, but it is still close enough to be considered part of the drawing, like in Figure 15);
- It can't be added.

When the new `Segment` cannot be added to the current `MultipleSegments` (or there is none), the current `MultipleSegments` is ready to be recognized and converted to an element (as will be show in Section 4.2) and a new `MultipleSegments` is created with the new `Segment`.

Another way that the current `MultipleSegments` may enter the recognition process is when the user changes the current presentation unit or takes too long between strokes, since we consider that the user finished his/her drawing after a certain amount of time of inactivity.

4.2 Shapes

In order for the recognition process to take place, the user drawings (already converted to `Segment` and `MultipleSegments`) must first be associated to a known shape. In UISKEI, a **`Shape`** is defined by the series of stroke directions that makes its drawing, similar to the work of (Cha, Shin, & Srihari, 1999). The directions can be one of the four cardinal directions (N, S, E, W) or the four ordinal directions (NE, NW, SE, SW). To simplify the notation, each direction was associated with a single character, as shown in the following figure:

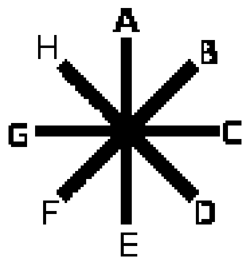

Figure 16: The directions compass rose.

Based on this abstraction, each shape of `p` points can be denoted as a string of `p-1` characters. But a shape can be drawn in different ways and the application should respond to how the drawing looks, not how it was made (Sezgin, Stahovich, & Davis, 2006). Taking for example the rectangle, the drawing has 5 points (to close it, the end point should be in the same position as the first point), so it can be expressed as a string of 4 characters, as illustrated in the following figure:

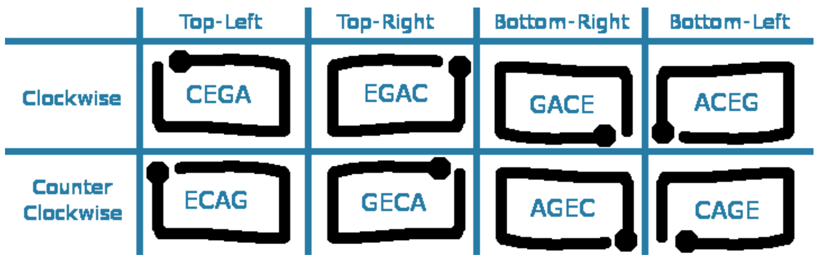

Figure 17: The rectangle shape as a string.

As can be seen, a shape can be expressed by different combinations of its `n` directions. If the shape is open (i.e. the end point is different from the initial point), like a '`\_|`' shape, it has 2 variants: drawing from left to right ("DCA") or drawing from right to left ("EGH"). If the shape is closed, as in the rectangle example, it depends on the starting point of the drawing (the columns in Figure 17), having `n` possibilities, and the direction of the drawing (the rows in Figure 17), having 2 variations (clockwise and counter-clockwise) to each previous possibility, summing up to `2n` variations.

When a user wants to create a new shape, he/she must create a text file with only the "base case" of the shape and whether it is closed or not, name the file with the name of the shape and with a `.shp` extension and place it in a specific directory. The text should follow this pattern, where the first term indicates if it is a closed shape (`y`) or not (`n`) and the second term is the string of the "base case":

```
[y/n] ([A,B,C,D,E,F,G,H]+)
```

UISKEI will look for available shapes when building the shape library at load time, creating all the variations in the process. For example, the `Rectangle.shp` file should contain only the line

```
y CEGA
```

and UISKEI would generate the 8 possible strings ("CEGA"/"ECAG", "ACEG"/"GECA", "GACE"/"AGEC", "EGAC"/"CAGE").

4.3 Element descriptors

Besides a list of `Shapes`, the recognition process also needs a list of `ElementDescriptors`. An `ElementDescriptor` defines many characteristics of a recognized widget, such as how it is drawn, what its possible states are (to be described later, in Section 4.5), how it handles events, which events it handles, and also how the elements can be recognized.

By having a descriptor, the behavior of an element is delegated to it, allowing the customization of the known widgets. Ideally, the user should be able to implement new widgets by creating new `ElementDescriptors`. However, the definition of a description language to allow the customization of widgets, such as described in (Hammond & Davis, 2006), lies outside the scope of this dissertation. Consequently, the `ElementDescriptors` were hard-coded, allowing for easier manipulation during the development phase and shedding some light on which operations and operators the descriptor language should support.

At the moment, UISKEI supports the following elements:

- Button
- Checkbox
- DropDown
- Frame
- Label
- Radio
- Spinbox
- Textbox

Since the creation of the elements is done through drawing, the `ElementDescriptor` should know how the element is drawn. In order to do that, it uses the `Shape` name and may apply some restrictions to it, such as a limit in height and/or width and whether it was drawn inside a specific element. The last restriction is responsible for the "evolution" of elements, a characteristic unique to UISKEI among the researched tools. A summary of how the widgets are created is seen in Figure 18:

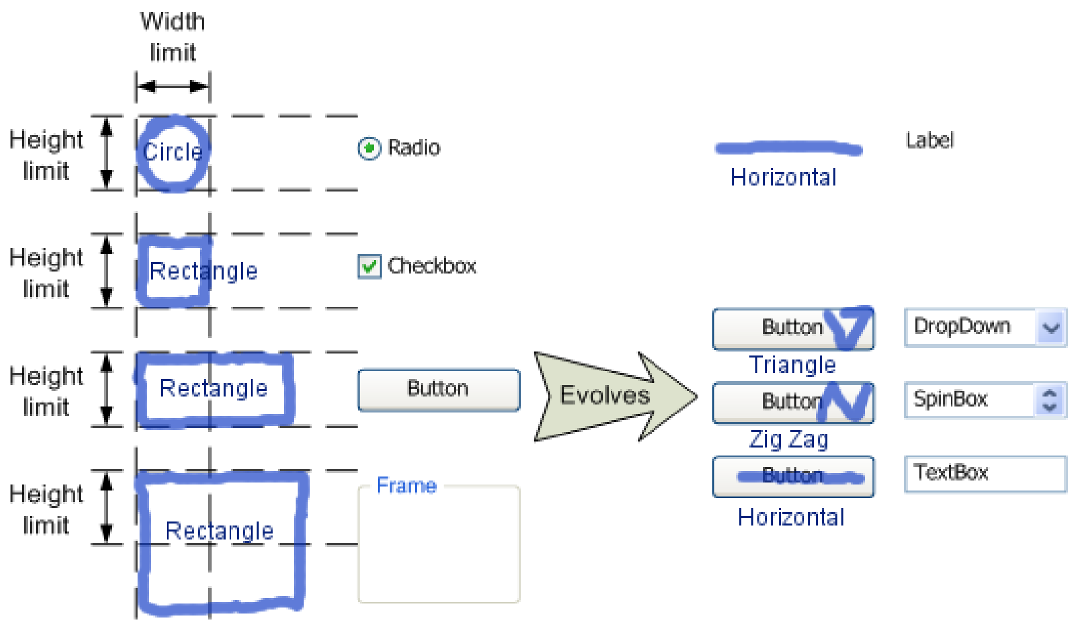

Figure 18: Language to create elements.

While in recognition mode, a small circle will be converted into a radio button and a small square, to a checkbox, for example. Figure 18 shows how both restrictions work: the size restriction is what determines if the rectangle is a checkbox, a button or a frame, and the evolution restriction determines if the horizontal line is a label or a textbox.

4.4 Recognition

When a `MultipleSegments` enters the recognition process, each of its `Segments` is simplified and converted into a string, following the same notation described in Section 4.2. The simplification process uses the Douglas-Peucker algorithm (Douglas & Peucker, 1973) to find the drawing's significant points, reducing noise as can be seen in Figure 19, and a tolerance regarding the size of which line segments should be converted into a "direction character".

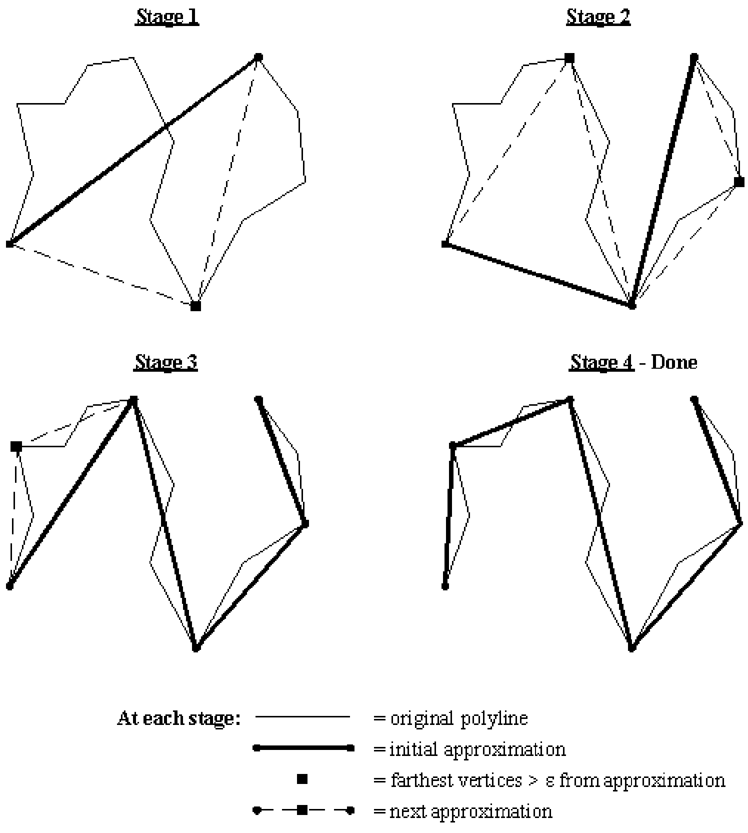

Figure 19: Douglas Peucker algorithm[14]

[14] Image taken from: http://softsurfer.com/Archive/algorithm_0205/algorithm_0205.htm#Douglas-Peucker Algorithm

Then, this string is compared to all strings corresponding to each preloaded shape using the Levenshtein distance algorithm (Gusfield, 1997, p. 215), using the difference between directions as a cost to the algorithm's operation. The overall distance between the `Segment`'s string and the shape's string is calculated proportionally to the segment's length, so that longer `Segments`, more prone to noise and less accurate, can still be recognized.

If the smallest distance (i.e., the best match) is less than a threshold, the `Segment` is associated to the `Shape`. If an association is made, it runs through the list of loaded `ElementDescriptors` to check for the first possible match. For the elements' evolution to work, the list should be ordered by the descriptor complexity: if it does not evolve, its complexity is 1; otherwise its complexity is 1 + the complexity of the "ancestor" element. This guarantees that the most complex elements will be checked first, ensuring that, for example, a horizontal line will generate a label only if it is not possible to generate a textbox with it.

If no association to a `Shape` is made or no `ElementDescriptor` matches the configuration, the `Segment` remains as it was drawn (an unrecognized element is called a `Scribble`). This way, the user can create and manipulate any kind of new element, even if it is not turned into a widget, making the software more flexible.

4.5 Element properties

Each element, no matter whether recognized, unrecognized, or a group, has a number of properties which will be defined in the next sections.

- **Name**: The element's name is how the designer will reference the element throughout the design process. By default, each element will be created with a name in the form `<type of element><id>` (e.g.: Button1, Checkbox4, Radio42), assigning a unique name to each new one in the same presentation unit (elements from different presentation units may have the same name).

- **Label**: The label is an auxiliary text that accompanies the element and varies its position accordingly, as can be seen in Figure 20 (the names of the elements were written as their labels):

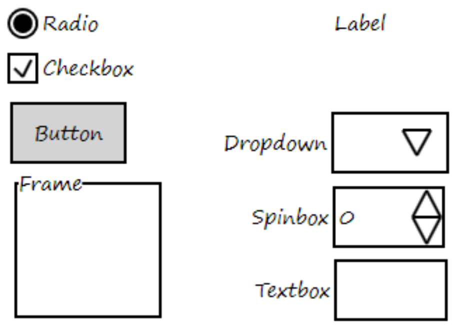

Figure 20: Labels.

- **Position**: The position `(x,y)` in the presentation unit, always referring to the top-left position of the element related to the top-left of the presentation unit, disregarding the element's label.
- **Size**: The element's width and height, disregarding the element's label dimensions. Some elements are created with fixed values in either direction, regardless of their drawing. For example, checkboxes are always created with the same width and height, while buttons are always created with the same height to help to establish a more consistent look and feel.
- **Enabled / Disabled**: This relates to how the element is first displayed during a simulation with the client. When the element is disabled, its representation is grayed, so the designer can know what the initial configuration is.
- **Visible / Invisible**: Similar to enabled/disabled, indicates if the element is visible in the beginning of the simulation. If the element is invisible, it is drawn in a light shade of blue in the "design view" and it does not appear to the client during the simulation. If the element is disabled and invisible, the representation of invisible is the one shown.
- **States**: States are possible values for the elements. Each one has its own set, expressed in the following table:

Table 2: Elements' states.

Element	Number of states	States
Button	No states	
Checkbox	3 states	Checked / Unchecked / Mixed
DropDown	Custom states	A string that the user can select later
Frame	No states	
Label	No states	
Radio	2 states	Checked / Unchecked
Spinbox	2 states	Up / Down
Textbox	3 pre-defined + Custom states	Blank / Valid / Invalid A name, a pattern and a sample text

In the textbox case, the states are defined in a table-like fashion. Each one has a name, which is how the state is presented to the designer, may have a pattern, which is used in the simulation to propose the state after the user's input of text, and may have a default text, which is the text to appear at the initial state. The pattern is a regular expression, which will be used when a text is entered in the textbox to determine its new state.

- **Initial state**: The state shown to the user at the beginning of the simulation.