Dynamic Query Interface for Spatial Proximity Query with Degree-of-Interest Varied by Distance to Query Point

Myoungsu Cho1, Bohyoung Kim1, Dong Kyung Jeong2, Yeong-Gil Shin1, Jinwook Seo1
1School of Computer Science and Engineering
Seoul National University
599 Gwanak-ro, Gwanak-gu, Seoul 151-744, Korea
msdanbi@hcil.snu.ac.kr,
{bhkim, yshin, jwseo}@cse.snu.ac.kr
2Samsung Electronics SAIT
San 14-1, Nongseo-dong, Giheung-gu, Yongin-si,
Gyeonggi-do 446-712, Korea
dongkyun.jeong@samsung.com

ABSTRACT
In this paper we present an interactive query interface called “TrapezoidBox” to support spatial proximity queries where users’ degree of interest varies depending upon the degree of separation from the point of interest. Spatial proximity queries are commonly built in information seeking tasks especially on online maps. If not impossible, it is hard to formulate spatial proximity queries using existing dynamic query widgets such as range sliders. TrapezoidBox allows users to easily build spatial proximity queries by interactively adjusting a trapezoidal function. Our controlled user study results show that TrapezoidBox has several advantages over a baseline interface with range sliders.

Author Keywords
Dynamic query interface, spatial proximity query, degree-of-interest, tab interface, visual information seeking.

ACM Classification Keywords
H5.2. Information interfaces and presentation (e.g., HCI): User Interfaces.

General Terms
Design, Human Factors.

INTRODUCTION
Dynamic queries are effective tools for interactively querying large datasets (e.g. databases), which can satisfy users’ highly diverse needs for data filtering [14]. Even though their expressive power is weaker than conventional database query languages such as SQL, their visual nature and their support of rapid, incremental, and reversible actions enable users to interactively identify global trends and to find answers to specific questions [15].

The dynamic queries are formulated using widgets such as sliders, buttons, and check boxes, each of which specifies satisfying conditions for corresponding attributes of a database. The query condition specified by a widget such as sliders is applied to the whole dataset (i.e., not to a specific subset of the dataset). Within each slider, the satisfying values are disjunctive, but conditions specified by a set of sliders are conjunctive. This kind of simple conjunction of global disjunctive queries can help users answer various specific questions by repeatedly performing such dynamic queries [14].

There also have been attempts to improve the expressive power of dynamic queries to support more general and complex queries. For example, Young and Shneiderman's innovative water flowing metaphor [18] can implement a complete set of Boolean expressions. However, there are still practically important queries that cannot be easily formulated by conventional dynamic query interfaces such as sliders. One of them is a complex query in which people specify the degree of interest differently depending on the distance from or similarity to a current query point of interest. The followings are two scenarios where people want to perform such complex queries:

(1) Assume that a researcher is staying at Hyatt Regency Atlanta for CHI 2010 conference. He/She wants to find a good restaurant for dinner. He/She would like to go to a restaurant within 300 meters walking distance if its customer rating is 3 or above. He/She is also willing to go to a restaurant farther than 300 meters (but no farther than 1 km) if its rating is better than 3, but his/her expectation increases as it gets farther than 300 meters.

(2) Assume that a tourist is in Rome. He/She is interested in antique architectures, especially built before A.D. 1700. He/She is willing to go see buildings built before A.D. 1700 if they are within 500 meters walking distance, but he/she is also willing to go a bit farther (but no farther than 1 km) if the buildings are much older. Again, his/her expectation increases as it gets farther than 500 meters.

These situations are not uncommon. When people perform an information search task, they often have to consider more than one closely related factor simultaneously. Moreover, they often want to adjust the satisfying range of one factor according to the value of the other factor as shown in the two given scenarios. For example, the rating...
of a restaurant that the researcher is willing to go to should be higher if it is located farther than a certain distance, and more specifically, the rating threshold usually gets gradually higher as its distance from the researcher gets longer.

Mounting and MacFarlane spelled out two important assumptions about how information can be more relevant to the context of the user: *spatial proximity* and *temporal proximity* [10]. They said that *spatial proximity* can be represented with a surface that can differentiate according to the *degree of separation* from the user’s current location. *Temporal proximity* can be explained in a way that regions that can be reached sooner are considered more relevant than those that are “temporally distant.” In this paper, we present a novel query interface to intuitively support the interactive formation of spatial proximity queries.

The formulation of queries for the abovementioned scenarios using current dynamic query interfaces is hard, if not impossible. To support it, it is necessary to do a series of disjunctive queries with an attribute (for example, distance for (1) and (2)). A tricky part is that the threshold value for the other attribute (customer rating for (1) and built year for (2)) should change according to the first attribute’s value. A makeshift solution to this problem based on conventional dynamic query interfaces could be to use a disjunction of a series of conjunctions using the two attributes. Each conjunction specifies a query condition for a specific range of values for an attribute and a corresponding value of the other attribute.

This makeshift solution might frustrate users when they are asked to formulate the spatial proximity queries (i.e., disjunction of conjunctions) because they are used to a conjunct of disjuncts while using tools and software with popular traditional dynamic query widgets. To address this problem, we present a novel dynamic query interface entitled TrapezoidBox (Figure 1) to intuitively support the query formation tasks for the scenarios depicted in (1) and (2), where spatial proximity queries are necessary.

TrapezoidBox aligns an attribute horizontally and the other attribute vertically. Threshold values are specified as a
Trapezoid-shaped function. For example, for scenario (1), the distance attribute is represented as the horizontal axis and the customer rating attribute is represented as the vertical axis. Users can drag the four vertices and three edges of the trapezoid horizontally and/or vertically to change the query condition. It is important to note that the utility of TrapezoidBox query interface is not limited to spatial proximity queries. It can be used to formulate composite queries where two quantitative attributes are involved and users want to specify query conditions where the satisfying range of one attribute is affected by the other attribute. We will present more general examples when we explain TrapezoidBox interface in detail.

We performed a comparative controlled user study to evaluate our novel dynamic query interface against a possible conventional method using traditional range sliders. We learned that our new interface is faster and more accurate in formulating spatial proximity queries.

This paper is organized as follows. After describing closely related work, we describe a new dynamic query interface, TrapezoidBox in terms of visual representation and user interaction. We then present what we learned from our controlled user study. After discussion on how to organize multiple dynamic query results using TrapezoidBox, we close this paper with plans for future work and conclusions.

RELATED WORK

Dynamic queries are a well-known successful approach to information seeking that can allow users to deal with information overload by interactively formulating query and rapidly showing the results [3, 14]. Many information visualization research and commercial systems adopted dynamic queries as an essential component.

Dynamic HomeFinder [17] was one of the first example systems that implemented dynamic queries. It helps users explore a real-estate database and find satisfying homes. Dynamap [12] also implemented traditional dynamic queries to help users find trends in Census data overlaid on a 2D map. PhotoFinder allows users to specify multiple disjunctive range selections using multiple two-sided slider bars to show a histogram chart for each attribute [9]. Filmfinder [13] supports users to search for certain films in large film database using various dynamic query devices. Spotfire [2] is the first commercially successful information visualization tool that is based on traditional dynamic queries. These applications and tools implement traditional dynamic query widgets and support only conjunct of disjuncts [15]. Spatial proximity queries need dynamic queries that support disjunct of conjuncts.

Various interface widgets such as range sliders, check boxes, and buttons are used to formulate traditional dynamic queries. There also have been a few approaches to extend the expressive power of traditional dynamic queries. AlphaSlider [1, 11] is a query device to allow users to select one item from an alphabetically arranged list without using a keyboard. Timebox widget of TimeSearcher [8] enables users to draw multiple rectangular boxes to specify conjunctive queries and find the time series data with a certain pattern. QuerySketch [16] allows users to draw a freehand line graph and find data that match the graph.

Young and Shneiderman also presented a novel water flowing model to support a complete set of Boolean expressions [18]. There also have been some approaches to allow users to perform more general Boolean queries using the dynamic query technique. Fishkin [7] presented a new dynamic query interface to support compound queries by overlapping multiple filters such as Magic Lens filters [5]. Users can formulate queries of various combinations such as disjunction of conjunctions as well as conjunction of disjunctions. This work is most similar to our approach, but it was not design to support the formation of the spatial proximity query where the degree-of-interest varies (gradually increased/decreased) depending on the distance to the query point.

Recently, many researches on filtering geographic data on the map have been done, especially on mobile maps. MAGDA [6] is a dynamic query system for geographic data on mobile devices and it is focused on the visualizations of query results. WebPark search interface [10] implements the ‘search around me’ filter to support spatial proximity queries within a restricted region of interest according to a specified distance from user’s current location. But, users can define only one condition for one specified distance. A mash-up mobile map application was also presented, where users can perform filtering by users’ profile and time information as well as by user’s position [4].

TRAPEZOIDBOX

Interface Design

TrapezoidBox query interface is a trapezoid-shaped widget designed for supporting formulation of spatial proximity queries where the degree of interest of an attribute differs according to the distance from a current query point of interest (Figure 2). Using TrapezoidBox, users can dynamically filter data when two attributes relate to each other and the satisfying range, or the threshold of one attribute is affected by the distance.

While letting d be an attribute for the distance from the current query point, the horizontal axis represents the distance d and does the vertical axis the other attribute. TrapezoidBox specifies a composite query involving the two attributes as a function \( f(d) \) associated with two functions \( f_1(d) \) and \( f_2(d) \) as shown in equation 1 (Figure 2a).

\[
 f(d) = \begin{cases} 
 f_1(d), & 0 \leq d \leq d_1 \\
 f_2(d), & d_1 < d \leq d_2 
\end{cases}
\]

eq 1
When a location is within $d_1$, the vertical attribute has a constant threshold of $f_1(0)$, indicating that the attribute has the fixed range of $[y_0 \sim f_1(0)]$. When a location is between $d_1$ and $d_2$, the threshold of the vertical attribute varies linearly as the distance increases from $d_1$ to $d_2$, indicating that the threshold becomes more stringent gradually. The satisfying data, $S$, for the composite query $f(d)$ is defined by the following equations:

$$S = \{ \text{data} | (\text{dist(data)} \leq d_1 \land \text{attr_val(data)} \geq f_1(\text{dist(data)}))$$

$$\lor (d_1 < \text{dist(data)} \leq d_2 \land \text{attr_val(data)} \geq f_1(\text{dist(data)})\} \ldots \text{eq. 2.1}$$

$$S = \{ \text{data} | (\text{dist(data)} \leq d_1 \land \text{attr_val(data)} \leq f_1(\text{dist(data)}))$$

$$\lor (d_1 < \text{dist(data)} \leq d_2 \land \text{attr_val(data)} \leq f_1(\text{dist(data)})\} \ldots \text{eq. 2.2}$$

where $\text{dist()}$ is a function returning the value of the distance attribute for a data and $\text{attr_val()}$ is a function returning the value of other related attribute for a data. Equation 2.1 is for a downward increasing vertical axis and equation 2.2 is for an upward increasing vertical axis.

In spatial proximity queries in which the distance attribute is typically represented as the horizontal axis, the trapezoid is reflection-symmetric against a vertical line at the current query point. Depending on the task, the vertical axis is upward increasing or downward increasing, but the filtering threshold always becomes more stringent downward. In case of scenario (1), the query finds restaurants whose customer ratings are higher than a threshold rating varied by the distance, thus the vertical axis for the customer rating is downward increasing (Figure 2b). In case of scenario (2), the query finds buildings which were built before a threshold year varied by the distance (that is, finding buildings whose built year is less than a threshold year), thus the vertical axis for the built year is upward increasing (Figure 2c).

TrapezoidBox can be used to formulate more general composite queries, not limited to spatial proximity queries. In other words, the horizontal axis of TrapezoidBox can represent any quantitative attributes such as date, time, or price. When a query function relates such attributes to another attribute for the vertical axis, the TrapezoidBox interface may have an asymmetric shape. We list three more general examples of composite queries along with corresponding TrapezoidBox widget sketches in Table 1.

### Interaction

TrapezoidBox has four vertices (Figure 3). Two ending vertices lying on the horizontal line, $v_1$ and $v_2$, specify the range of the attribute represented as the horizontal axis. The three edges, $v_1v_2$, $v_2v_3$, and $v_3v_4$ specify the threshold of the attribute for the vertical axis, which functionally relates to the horizontal-axis attribute. The convex region surrounded by the three edges and the horizontal axis corresponds to the composite query conditions of the two related attributes, and the area of the convex region is proportional to the size of the satisfying result set.

A mouse-over on a vertex point changes the cursor to a hand cursor, indicating that this vertex can be moved. When

\[ \text{Table 1. Examples of composite queries using TrapezoidBox} \]

<table>
<thead>
<tr>
<th>Task</th>
<th>TrapezoidBox</th>
</tr>
</thead>
<tbody>
<tr>
<td>A user needs to find some files from his/her computer which were likely to be created during a specific period and were more likely to be created during a narrower period.</td>
<td>![Diagram of TrapezoidBox with labeled vertices and edges]</td>
</tr>
<tr>
<td>A user wants to buy a high-quality product at a good price. If it costs less than specific dollar amount, the quality better than a threshold is fine, but as the product becomes more expensive, the quality should get better.</td>
<td>![Diagram of TrapezoidBox with labeled vertices and edges]</td>
</tr>
<tr>
<td>A user wants to buy a low mileage used car. If the car is young, the mileage less than a threshold is fine, but as the car gets older, the mileage should get lower.</td>
<td>![Diagram of TrapezoidBox with labeled vertices and edges]</td>
</tr>
</tbody>
</table>
users click on and drag a vertex point horizontally or vertically, the values of corresponding attributes are adjusted. The two ending vertices, \( v_1 \) and \( v_4 \), can move only left and right. The remaining two vertices, \( v_2 \) and \( v_3 \), can move up and down as well as left and right; however, neither they cross over each other nor move over the two ending vertices, \( v_1 \) and \( v_4 \).

A mouse-over on one of the three edges changes the cursor to an arrow cursor, indicating that the edge can be moved. Users can drag up and down the top edge (\( v_2 v_3 \)), adjusting the threshold of the vertical-axis attribute. Users can also move the two side edges (\( v_1 v_2 \) and \( v_3 v_4 \)) left or right. When the positions of the vertices or the edges are modified, the satisfying query result is updated immediately. In case of the symmetric TrapezoidBox for spatial proximity queries, the change of every vertex point or edge position in one side triggers a symmetric change in the other side of the widget.

**CONTROLLED USER STUDY**

We conducted a controlled user study to evaluate TrapezoidBox query interface in terms of how well people could learn and use it in formulating spatial proximity queries, compared to an extension of conventional range sliders.

**Baseline Interface: a set of range slider pairs**

To comparatively evaluate TrapezoidBox, we implemented a baseline interface using conventional range sliders. To adjust the range of two related attributes simultaneously (e.g., the distance and customer rating in scenario (1)), a pair of range sliders is required, one slider for each attribute. A pair of range sliders can represent a single conjunctive query with two corresponding attributes. When applied to formulating spatial proximity queries in which the threshold for an attribute changes depending upon the degree of separation from the current query point, more than one slider pair is needed. For accurate formation of a query representing a gradual changing relation between two attributes as in scenario (1), too many slider pairs are required depending upon the desired query granularity; however, it is impractical in real querying situations thus we limited the number of slider pairs to 5.

Figure 4 shows an example of formulating a disjunction of a series of conjunctive queries using \( \leq 5 \) slider pairs for task 4 in the user study described later (or refer to the task description displayed to the right of the map). The upper range slider in a slider pair represents the distance. It has two bars and the in-between region is highlighted with an orange bold line. The range sliders for the distance gradually shift to the right while moving over the next slider pair. The satisfying distance range is represented as a circular region in the map below. The lower range slider representing the customer rating also has two bars. It has a range value which gradually shortens, anchored at the right bar.

At first, a single slider pair is provided and more slider pairs can be added as necessary up to 5. Each pair of range
sliders is surrounded by a gray box indicating that the two range sliders in it are closely related to represent a single conjunctive query condition. If users click on the “Add” button on the top right corner, a new pair of range sliders is added. Users can delete a slider pair by clicking on the ‘x’ button located on the top right of the surrounding box. The changes are updated in real-time and the satisfying query results are displayed immediately on the map.

Participants
12 (5 females) volunteers from Seoul National University participated in the study. We screened participants for color blindness and required normal or corrected-to-normal eyesight. The average age of participants was 30 (31.6 for males, 27 for females), ranging from 21 to 38 years of age. All participants had a prior experience in using a map on desktop or mobile environment. Their usage frequency was 5 to 6 times per month. The experiment took about 20 minutes and participants were rewarded about $10. Participants were also told at the beginning of the session that the best performing participant could win an additional prize (a USB flash memory).

Tasks and Materials
Participants were asked to imagine that they were traveling to New York and Atlanta. Their trip plan is to visit old buildings in New York, and then go to famous restaurants in Atlanta. Information about buildings in New York was collected by searching Wikipedia with a keyword “list of the oldest buildings in New York.” Name of a building, its first built year, and its location were collected. Information about restaurants was acquired by searching Google with a keyword “restaurants in Atlanta.” Name of a restaurant, its address, and its customer rating were collected and its location was obtained by translating their addresses to the location on the map.

Participants performed four tasks with two different levels of complexity (2 simple tasks and 2 complex tasks). They were asked to finish these tasks as fast and precisely as they could. Tasks are shown below:

1. Find buildings built before 1950's within 300 meters.
2. Find restaurants with four stars or above of customer rating within 800 meters.
3. Find buildings built before 1950's within 300 meters, and buildings within 300 meters to 1 km but with progressively older buildings at a farther distance.
4. Find restaurants with two stars or above of customer rating within 300 meters, and restaurants within 300 to 900 meters with progressively higher ratings for the ones at a farther distance.

Tasks 1 and 2 are simple tasks and tasks 3 and 4 are complex ones. These tasks were designed to test how well TrapezoidBox can support users’ query formation. All four tasks required only symmetric TrapezoidBox widgets. When running the complex tasks with the baseline interface, the number of slider pairs needed was clearly stated to participants to help them formulating the query. Otherwise, it was often impossible for people to figure out how to formulate queries for the complex tasks using the baseline interface. In a pilot study with 3 subjects, all of them could not finish the complex query tasks without such the clue.

Study Design and Procedure
We ran the study as a within-subjects design, with each participant performing all the tasks using all the interfaces. Before the experiment, participants were given tutorial on the experiment. Before beginning the test of each interface, participants also performed a representative practice task in order to familiarize themselves both with the task and interface. As a practice dataset, restaurants at local regions were used because they were familiar with the participants.

We ran the experiment as a 2 (Interface: baseline, TrapezoidBox) × 2 (Task Complexity: simple, complex) × 2 (Question) design while counterbalancing the order of interfaces to avoid the learning effect. The four task questions (2 complexity × 2 questions) were always presented in the order from simple tasks to complex tasks, more specifically in the fixed order from task 1 to task 4. For each interface, we used a different location as the query point.

When performing a query task, participants first understood the task appearing on the right pane in study session management software (Figure 4), and then they pressed the “Start” button when they were ready. Then, for the baseline interface, a pair of sliders is displayed with each slider having an initial range from minimum to maximum values for the corresponding attribute. For the TrapezoidBox interface, a TrapezoidBox interface widget initially has the same ranges as for the baseline interface, thus it is initially rectangle-shaped.

Participants were asked to click on the farthest data-point before pressing the “Finish” button because we wanted to simulate a realistic situation in which people usually check on the query result. Even when a data-point other than the farthest one was clicked on, it was not considered as an error as long as the query conditions were correctly set. Participants were asked to finish each task as quickly and accurately as possible. At the end of a study session, participants filled in a questionnaire for subjective evaluation. The experiment took about 25 minutes for each participant.

Equipment
We ran twelve participants performing tasks independently in each session of the study. Each participant worked on an Intel core i7-920 processor (2.67GHz) and 4GB memory PC with a single 22" Dell LCD display running at a 1680 × 1050 pixel resolution. Study session management software running on the computer drove all tasks and logged results. For all the tasks, the software collected task time and
correctness for each question. We used these metrics to estimate the efficacy of each interface. The study session management software was developed in C# and JavaScript using the Google map API.

Hypotheses
We established two hypotheses of this evaluation study.
(1) In case of simple tasks, two interfaces will show similar performance in the task completion time and error rate.
(2) In case of complex tasks, TrapezoidBox will outperform the baseline interface in both the task completion time and error rate.

Statistical Analysis and Results
We measured and analyzed three parameters from the experiment: total elapsed time to complete a task (time between pressing the “start” and “finish” buttons); total length of mouse dragging per task; and error. We also performed statistical analysis of questionnaire responses.

Task Completion time
We analyzed the task completion time with a 2 (Interface) \times 2 (Task Complexity) repeated-measures analysis of variance (RM-ANOVA). We found a significant main effect of Interface \((F_{1, 92} = 30.4, p < .0001)\), indicating that participants completed a task in a significantly less time with TrapezoidBox than with the baseline interface (Figure 5a). We also found a significant main effect of Task Complexity \((F_{1, 92} = 59.9, p < .0001)\) (Figure 5a). It was trivial that the complex tasks took more time to complete than did the simple tasks.

An interaction was found between Interface and Task Complexity \((F_{1, 92} = 28.81, p < .0001)\) (Figure 5b). When moving from simple tasks to complex tasks, the task completion time drastically increased for the baseline interface; however, TrapezoidBox showed a relatively slow increase in the task completion time. A likely reason for this might be that upon the complex tasks, participants had to add slider pairs as needed and then adjust the slider range on an individual slider while formulating a given complex query in their brain. By contrast, they could represent a dynamic query just by dragging vertices or edges in a single trapezoid using TrapezoidBox. It was observed that the participants easily adapted to TrapezoidBox.

For each task type, we compared the task completion time between the baseline interface and TrapezoidBox using RM-ANOVA. For the simple tasks, the task time was not significantly different between the two interfaces \((F_{1, 24} = 0.15, p = .70)\); however, for the complex task, TrapezoidBox took significantly less time to complete a task than did the baseline interface \((F_{1, 24} = 47.81, p < .0001)\) (Figure 6). This result could be similarly explained as in the interaction between Interface and Task Complexity.

Total length of dragging to complete a task
The length of mouse dragging on each interface was recorded in pixels. We analyzed the drag length with a 2 (Interface) \times 2 (Task Complexity) RM-ANOVA. We found a significant main effect of Interface \((F_{1, 92} = 73.39, p < .0001)\), indicating that participants completed a task using significantly shorter mouse dragging than with the baseline interface (Figure 7a). We also found a significant main effect of Task Complexity \((F_{1, 92} = 70.68, p < .0001)\) (Figure 7a). It was trivial that the complex tasks required much more frequent mouse dragging, more specifically in the baseline interface with many range sliders.

An interaction was found between Interface and Task Complexity \((F_{1, 92} = 54.09, p < .0001)\) (Figure 7b). When moving from simple tasks to complex tasks, the drag length drastically increased for the baseline interface; however, TrapezoidBox exhibited a very slow increase in the drag
length. A likely reason for this might be that upon the complex tasks, participants had to drag many slider bars in additional slider pairs when using the baseline interface. On the contrary, they were able to represent the complex queries easily by dragging vertices a little bit in TrapezoidBox.

For each task type, we compared the drag length between the two interfaces using RM-ANOVA. The participants completed a task with significantly shorter mouse dragging using TrapezoidBox than using the baseline interface, for both simple and complex tasks ($F_{1, 24} = 28.93$, $p < .0001$; and $F_{1, 24} = 124.42$, $p < .0001$, respectively) (Figure 8). It was easily expected that with the complex tasks, the baseline interface needs to drag many range bars in multiple sliders, up to 10 (5 slider pairs $\times$ 2) sliders. Even with the simple tasks, it turned out that TrapezoidBox outperformed the baseline interface.

**Error ratio**

All the participants did not make any errors with TrapezoidBox. When using the baseline interface, they also made no errors for the simple tasks; however, 7 out of 12 participants made errors for the complex tasks. We classified errors into the following three categories.

- **Redundancy error**: Let’s assume that there are two slider pairs A and B for scenario (1). If the range of distance in A is the same as in B but the value of customer rating in A is different from the value in B, the redundancy error happens.

- **Incorrect distance range error**: It occurs when a required range of distance is missed or the distance range is incorrectly set.

- **Incorrect threshold error**: It occurs when the threshold value (e.g., customer rating for scenario (1) and year for scenario (2)) is incorrectly set.

When performing the complex tasks with the baseline interface, about 38% of participants made the redundancy errors and about 21% of them made the incorrect threshold errors. None of the participants made any incorrect distance range errors.

We compared the error ratio for the complex tasks between the two interfaces using Friedman’s Chi-Square test. Participants made a significantly greater number of mistakes with the baseline interface than with TrapezoidBox ($\chi^2(1) = 9$, $p = .0011$). It is not surprising because it requires many more interactions for participants to accurately adjust many sliders in the baseline interface.

**Subjective data**

Participants were asked to answer the following 6 questions to collect subjective ratings about the two interfaces by using a 7 point Likert scale [Rating: 1 = Strongly disagree; 7 = Strongly agree]. The questions with significantly different ratings between the two interfaces were marked with an asterisk (*).

Q1. This interface is easy to learn.
Q2. This interface is easy to use.
Q3. This interface is fun. *
Q4. Overall, I like this interface. *
Q5. It is easy to understand queries created by this interface.
Q6. I would like to use this interface again. *

![Figure 8. Drag length for each task type. For both simple AND complex tasks, participants used significantly less dragging with TrapezoidBox than with the baseline interface.](image)

![Figure 9. Average Likert scale rating for the two interfaces using the scale of 1= Strongly disagree and 7=Strongly agree.](image)
We analyzed the subjective rating result using Friedman’s Chi-Square test and found the significant results for Q3, Q4, and Q6 (for Q3, $\chi^2(1) = 9$, $p = .0001$; for Q4, $\chi^2(1) = 10$, $p < .0001$; for Q6, $\chi^2(1) = 6.4$, $p = .0046$) (Figure 9). Thus, we can summarize the significant results as follows. First, participants enjoyed using TrapezoidBox more than the baseline interface. Second, they were more favorable toward TrapezoidBox. Third, they were in favor of TrapezoidBox when they have to choose one for future use. Overall, 9 out of 12 participants preferred Trapezoid query interface to the baseline interface.

According to the freeform comments, the baseline interface was familiar and easy to control, but it was frustrating when they performed the complex tasks. This was because they had to repeatedly modify threshold conditions to accomplish the complex tasks. Participants liked TrapezoidBox because they could easily formulate queries. They said that it was intuitive to express what they wanted using TrapezoidBox.

**DISCUSSION AND FUTURE WORK**

Results from our comparative evaluation strongly supported the hypotheses. As expected, there was no difference in task completion time for simple tasks, but TrapezoidBox was significantly faster in formulating complex queries. Furthermore, participants made significantly less mistakes with TrapezoidBox than with the baseline interface.

We attribute this to the fact that the interface design and interaction of TrapezoidBox better support spatial proximity queries with the degree of interest varied by the distance from and similarity to the point of interest. Another likely reason for the evaluation results is that it is very hard to formulate complex spatial proximity queries using the range sliders. This is supported by the fact that participants made significantly more mistakes with the baseline interface support.

In this paper, we evaluated only the symmetric TrapezoidBox. Inspired by the promising study result, we thought that it might be interesting to see how well people can formulate more complicated queries with asymmetric TrapezoidBox (as in Figure 3). We believe that our technique can be more widely applicable to many general problems (some shown in Table 1) where users want to dynamically change the satisfying range of one attribute depending on the value of another related attribute.

In this paper we did not look into how people perceive the rate of change encoded by the slope of the degree-of-interest function in TrapezoidBox. It might be interesting to perform a user study to examine how the angle of the slope affects users’ perception of change in query conditions.

Inspired by users’ comments that TrapezoidBox resembles tabs in the tab interface, we thought that it could be effective to organize spatial proximity query results using TrapezoidBox as tabs. This TrapezoidBox tab interface might be more useful on small screens. Users can perform several spatial proximity queries based on different points of interests, and assign each query result to a TrapezoidBox tab as shown in Figure 10a–10c. The color of a circle matches the color of the corresponding TrapezoidBox tab.
We built a prototype on a mobile phone running on Windows Mobile 6.1 to receive promising comments. It might be interesting to examine how well this TrapezoidBox tab interface can support users search and filtering tasks, especially on mobile devices.

CONCLUSION
We presented a novel dynamic query interface called “TrapezoidBox” for spatial proximity queries. We conducted a comparative user study to examine whether it could improve users’ performance for query formation tasks. We learned that participants perform the query formation tasks with TrapezoidBox more efficiently than with the baseline interface. Overall, users preferred TrapezoidBox to the baseline interface with traditional range sliders.

ACKNOWLEDGMENTS
We thank Prof. Johan Lim (Department of Statistics at Seoul National University) for his help with statistical analyses. We also appreciate our study participants for their time and comments. This work was supported by the Engineering Research Center of Excellence Program of Korea MEST/KOSEF (R11-2008-007-01002-0), KOSEF grant (No. 2009-0064949), and the Brain Korea 21 Project. The ICT at Seoul National University provided research facilities for this study.

REFERENCES