Visualization of Dynamic Spatial Data and Query Results Over Time in a GIS Using Animation [*]

Glenn S. Iwerks and Hanan Samet[†]
Computer Science Department, Center for Automation Research,
Institute for Advanced Computer Studies
University of Maryland, College Park, Maryland 20742
{iwerks,hjs}@cs.umd.edu

Abstract

Changes in spatial query results over time can be visualized using animation to rapidly step through past events and present them graphically to the user. This enables the user to visually detect patterns or trends over time. This paper presents several methods to build animations of query results to visualize changes in a dynamic spatial database over time.

Keywords: dynamic spatio-temporal data, visualization, animated cartography

1 Introduction

To help detect patterns or trends in spatial data over time, animation may be effective [10]. Alerters [2] or triggers [18] may be used to determine when a particular database state has occurred, but it may be desirable for the user to be aware of the events leading up to a particular situation. For example, a trigger can notify the user when vehicles enter a particular area of interest. When a trigger is fired the user knows that an event has occurred but does not know what led up to the event. In some cases it may be sufficient for the user to simply monitor the display as events occur, but if the amount of time between events is very long or very short, this may not be feasible. Depending on the situation, events may take hours, days, or even years to play out. This can make it difficult for the user to visually detect associations between events that occur far apart in time. One approach is to render spatio-temporal information in a static map [3, 10] but these can be confusing, and hard to read. Alternatively, the display output may be captured when data is processed and then sequenced into an animation. Using animation techniques, changes in spatial query results over time can be viewed, rapidly stepping through past events. This paper presents several methods to accomplish this result.

In this paper we address the display of 2D spatial features changing over time in a geographic information system (GIS). A GIS is a spatial database containing georeferenced data. A spatial database is defined here as a database in which spatial attributes, such as points, lines, and polygons can be stored using a relational data model. A table of related attributes is called a relation. A tuple in a relation is one instance of these related items. Base relations in a relational database are part of

[*]VISUAL'00, *pages 166-177,* Lyon, France, November 2000

[†]The support of the National Science Foundation under Grant EIA-99-00268 and IRI-97-12715 is gratefully acknowledged.

the relational data model. A view in a relational database is a query defined on base relations and made available to users as a virtual relation. A view may be materialized, or in other words, stored on disk so that the view need not be recomputed from the base relations each time it is used in further queries. Nevertheless, a materialized view must be updated when the base relations are modified by a transaction. Updating materialized views is also known as view maintenance [4]. A database transaction is a set of changes to a database state such that the database is left in a consistent state when the transaction is complete. If a transaction fails to complete, then the database reverts to the previous state just before the transaction began.

The remainder of this paper is organized as follows. Section 2 discusses background work in the area. Section 3 presents some algorithms for creating and viewing animations of spatial query results as they change over time. Concluding remarks and plans for future work are presented in Section 4.

2 Background

Most, if not all, previous applications of animation to this domain have been to visualize changes in base data rather than to database query results [3, 10, 17]. In general, most previous methods render spatial data in a bitmap. One bitmap is created for each discrete time step in a series. The bitmaps are then displayed in succession creating an animation, or in other words, animated maps. This is also known as animated cartography.

2.1 Animated Cartography

Visualization of georeferenced spatio-temporal data has been a topic of study for over 40 years [3]. One approach is to use static maps where temporal components are represented by different symbols or annotations on the map. Another approach is to use a chronological set of ordered maps to represent different states in time [11], sometimes known as strip maps. With the advent of more powerful computers and better graphics capabilities, animated maps are increasingly used. Animation is used in the presentation of meteorological data in weather forecast presentations to show changes over time [17]. Animated cartography is also used for decision support in disease control to visually detect patterns and relationships in time-series georeferenced health statistics [14, 13]. Animation is used in the study of remote sensing time-series data [15, 16]. In [9] animated cartography is used in the presentation of urban environmental soundscape information for environmental decision support. The use of animation of spatio-temporal data in non-cartographic fields is presented in [8] and [12] to visualize dynamic scientific spatio-temporal data. The effectiveness of animation techniques to present time-series cartographic data to a user is studied in [10]. The study concluded that animation may be able to help decrease the amount of time needed for a user to comprehend time-series spatial data and to answer questions about it compared with other methods.

2.2 The Spatial Spreadsheet

The Spatial Spreadsheet [7] serves as a testbed for the algorithms presented in this paper (see Figure 1). It is a front end to a spatial relational database. In the classic spreadsheet paradigm, cell values are non-spatial data types whereas in the Spatial Spreadsheet, cell values are database relations. The purpose of the Spatial Spreadsheet is to combine the power of a spatial database with that of the spreadsheet. The advantages of a spreadsheet are the ability to organize data, to formulate operations on that data quickly through the use of row and column operations, and to propagate changes in the data throughout the system. The Spatial Spreadsheet is made up of a 2D array of cells. Each cell in the Spatial Spreadsheet can be referenced by the cell's location (row, column). A cell can contain two types of relations: a base relation or a query result. A query result is a materialized view [4] defined on the base relations. The user can pose a simple query in an empty cell. For instance, a cell might contain the result of a spatial join between base relations from two other cells. Simple queries are operations like selection, projection, join, spatial join [5], window [1], nearest neighbor [6], etc. Simple queries can be composed to create complex queries by using the result of one simple query as the input to another. If a base relation is updated, the effects of those changes are propagated to other cells by way of the query operators.

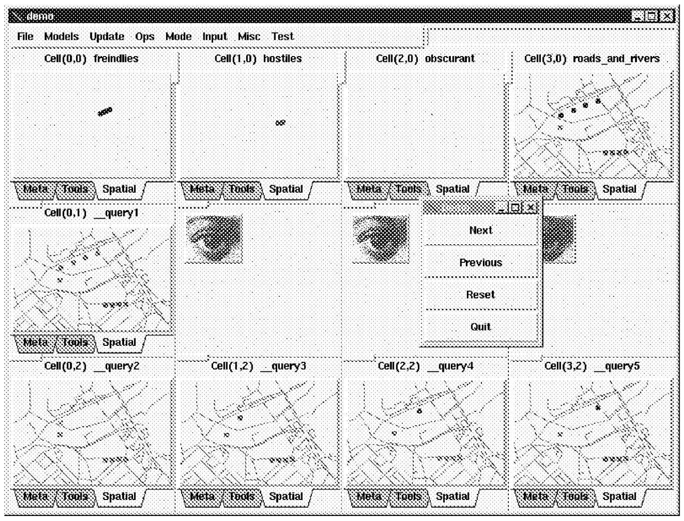

Figure 1: The Spatial Spreadsheet

3 Movie Mode

In the Spatial Spreadsheet, spatial attributes of a relation can be displayed graphically for each cell. When a base relation is updated, the change is propagated from cell to cell and the changes are reflected in each cell's display. To visualize changes over time, the display layers are saved and redisplayed in rapid succession like the frames of a movie. In the Spatial Spreadsheet, this is known as "movie mode". Each layer in the sequence is a movie animation frame. When contents of a base relation or a query result change, the old display layer is removed and saved, and then a new layer is rendered. When the user wants to play back the changes over time, layers are displayed successively in order from the oldest to the most recent.

3.1 Example Query

For the purpose of illustration, consider an example spatial join query in a spatial database. A join operation is a subset of the Cartesian product of two relations limited by a query predicate. The predicate is defined on the attributes of the two relations. For each tuple pair in the Cartesian product, if the query predicate is true, then the tuple pair is included in the result. A spatial join uses a query predicate defined on spatial attributes. An example spatial join query expressed in SQL is shown below.

```
SELECT *
FROM Observer, Target
WHERE Distance(Observer.Location, Target.Location) ≤ d
```

In this example, the schema of relations `Observer` and `Target` is (`Name`, `Location`) where `Name` is a string and `Location` is a point. The Distance() function returns the distance between two spatial attributes. The result of the query contains all the tuples joined from the two relations where attribute `Observer.Location` and attribute `Target.Location` are within distance d of each other.

Suppose at time t_0 relation `Observer` contains three tuples $\{(O1,(4,1)),\ (O2,(3,3)),\ (O3,(1,2))\}$, and `Target` has one tuple $\{(T1,(4,2))\}$ (see Figure 2a). Now consider the spatial join on these relations as expressed in the SQL query given above where d equals 1. The resulting output is shown in the first row of Figure 4 and graphically in Figure 3a. Now, suppose at time $t_0 + 1$ minute, the `Target` relation is updated by deleting tuple $(T1,(4,2))$ and inserting tuple $(T1,(3,2))$. The output of the recomputed spatial join is shown in the second row of Figure 4 and graphically in Figure 3b. Subsequently, suppose at time $t_0 + 4$ minutes the target at location $(3,2)$ moves to location $(2,2)$. The new spatial join output is shown in the third row of Figure 4 and graphically in Figure 3c.

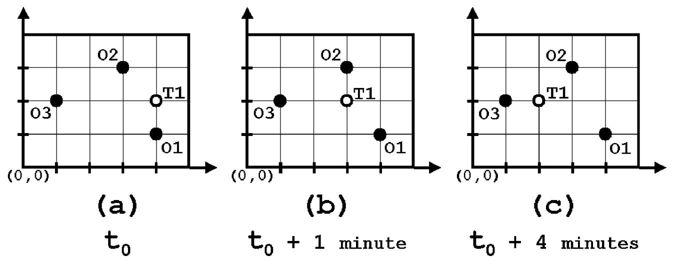

Figure 2: Graphical representation of spatial attributes in relations `Observer` and `Target` at different times. Point attributes of relation `Observer` are denoted by the • symbol and labeled Oi. Point attributes of relation `Target` are denoted by the ∘ symbol and labeled Ti.

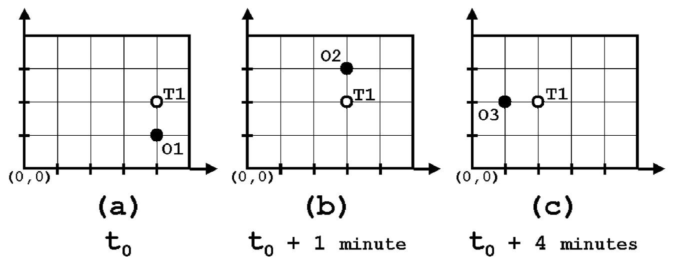

Figure 3: Graphical representation of the spatial join result between relations `Observer` and `Target` within a distance of 1 at different times. Point attributes of relation `Observer` are denoted by the • symbol and labeled Oi. Point attributes of relation `Target` are denoted by the ∘ symbol and labeled Ti.

3.2 Scan and Display

One can display results after a view is computed, or display results while the view is being computed. Function Scan_And_Display(), given below, is a simple function for rendering a frame from a materialized view, or base relation after an update. Input parameter R is a relation to be displayed.

Time	Name1	Location1	Name2	Location2
t_0	O1	(4,1)	T1	(4,2)
t_0+1	O2	(3,3)	T1	(3,2)
t_0+4	O3	(1,2)	T1	(2,2)

Figure 4: Spatial join result between relations `Observer` and `Target` within a distance of 1 at different times.

Parameter *movie* is a sequence of animation frames. Each call to Scan_And_Display() adds a new frame to the animation.

In line 1 Create_New_Frame() creates a new animation frame. The outer **foreach** loop scans relation R tuple-by-tuple. The inner **foreach** loop renders each spatial attribute in a tuple. Procedure Render() invoked in line 6 is an implementation-specific procedure which performs the mechanics of rendering a spatial feature into a movie animation frame. Render() may also display the current animation frame to the user as it is drawn. The '|' operator, used in line 9, appends the new frame to the movie sequence. The modified *movie* sequence is the return value of Scan_And_Display().

```
function Scan_And_Display($R, movie$):  return movie sequence
   begin
 1.   $movie\_frame \leftarrow$ Create_New_Frame()
 2.   foreach tuple $t$ in relation $R$ do
 3.      begin
 4.         foreach spatial attribute $a$ in tuple $t$ do
 5.            begin
 6.               Render($movie\_frame, a$)
 7.            end
 8.      end
 9.   $movie \leftarrow movie \mid movie\_frame$
10.   return $movie$
   end
```

3.3 Process Movie

Building an animation durring query processing adds processing overhead, but avoids rescanning the materialized view a second time. A movie frame is rendered durring processing using function Process_Movie() given below. Input parameter Q is a query to be processed. Parameter *movie* is a sequence of animation frames. Each call to Process_Movie() adds a new frame to the animation.

In line 1, a movie animation frame is created. Functions Process_First() and Process_Next(), in lines 2 and 9 respectively, process the query and return the next query result tuple. The **while** loop processes each query result tuple t until no more are generated. The **foreach** loop iterates through the spatial attributes in tuple t. Attributes are rendered in line 7. A new animation frame is appended to the sequence in line 11. The modified *movie* animation sequence is the return value of Process_Movie().

```
function Process_Movie($Q, movie$) :  return movie sequence
   begin
 1.   $movie\_frame \leftarrow$ Create_New_Frame()
 2.   $t \leftarrow$ Process_First($Q$)
 3.   while( $t \neq \emptyset$ ) do
```

```
        begin
          foreach spatial attribute $a$ in tuple $t$ do
            begin
              Render($movie\_frame, a$)
            end
          $t \leftarrow$ Process_Next($Q$)
        end
      $movie \leftarrow movie \mid movie\_frame$
      return $movie$
    end
```

3.4 Play Movie

Procedure Play_Movie(), given below, is used to play a movie created by either Scan_And_Display() or Process_Movie(). Parameter *movie* is a sequence of animation frames. The *frame_duration* input parameter controls the animation frame rate. The main loop iterates through the *movie* sequence and displays each frame. Procedure Show() is an implementation-specific procedure. It displays the *current_frame* to the user. Procedure Wait() halts the execution of Play_Movie() for the time period specified by *frame_duration*.

```
  procedure Play_Movie($movie, frame\_duration$)
    begin
      foreach $current\_frame$ in $movie$ do
        begin
          Show($current\_frame$)
          Wait($frame\_duration$)
        end
    end
```

3.5 Variable Update and Playback Rates

The algorithms presented so far work well if updates occur at regular intervals. If updates occur at random intervals, then the perception of temporal relationships between events may be distorted durring playback. This occurs because procedure Play_Movie() displays each frame for the same amount of time. Function Process_Variable_Rate(), given below, creates variable rate animations to support irregular time intervals between updates.

```
  function Process_Variable_Rate($Q, movie, transaction\_times$)
    : return two sequences
    begin
      $movie\_frame \leftarrow$ Create_New_Frame()
      $transaction\_times \leftarrow transaction\_times \mid$ Get_Last_Transaction_Time()
      $t \leftarrow$ Process_First($Q$)
      while($t \neq \emptyset$) do
        begin
          foreach spatial attribute $a$ in tuple $t$ do
            begin
              Render($movie\_frame, a$)
            end
          $t \leftarrow$ Process_Next($Q$)
        end
      $movie \leftarrow movie \mid movie\_frame$
```

```
13.    return $movie$ and $transaction\_times$
     end
```

Function Process_Variable_Rate() is similar to function Process_Movie(). One parameter is added and two lines are different. Parameter *transaction_times* is a sequence of numbers. Each element in *transaction_times* is associated with a *movie* animation frame representing the time at which a transaction took place resulting in the creation of the associated movie frame. The first element in *transaction_times* cooresponds to the first element in *movie*, and so forth. The time of the last transaction is returned by Get_Last_Transaction_Time() and is appended to the *transaction_times* sequence in line 2. The function return value is the modified *movie* sequence and the modified *transaction_times* sequence.

Procedure Play_Variable_Rate(), shown below, uses the *transaction_times* data gathered by Process_Variable_Rat to determine the duration of animation frames durring playback. At playback, the time between frames is proportional to the time between update transactions. As an example, consider the query given in Section 3.1. An update transaction occurs after one minute and the next one occurs after another three minutes. The resulting animation has three frames. If the animation is played back so that the duration of the first frame is 0.5 seconds, then it follows that the duration of the second frame is 1.5 seconds. At this rate, playback is 120 times faster than realtime. The input parameter *dilation_factor* controls playback rate. A value greater than 0 but less than 1 is faster than realtime. A value greater than 1 is slower than realtime. If *dilation_factor* $= 1$, then playback will be close to realtime plus some added time for processing overhead. The algorithm could be made more precise by subtracting the processing overhead time from the computed *frame_duration* value. For simplicity, processing overhead was not considered here.

```
   procedure Play_Variable_Rate($movie, transaction\_times, dilation\_factor$)
     begin
1.     $current\_time \leftarrow$ first($transaction\_times$)
2.     $transaction\_times \leftarrow$ rest($transaction\_times$)
3.     foreach $next\_time$ in $transaction\_times$ do
4.       begin
5.         $frame\_duration \leftarrow (next\_time - current\_time) * dilation\_factor$
6.         $current\_frame \leftarrow$ first($movie$)
7.         $movie \leftarrow$ rest($movie$)
8.         Show($current\_frame$)
9.         Wait($frame\_duration$)
10.        $current\_time \leftarrow next\_time$
11.      end
12     $current\_frame \leftarrow$ first($movie$)
13.    Show($current\_frame$)
     end
```

In procedure Play_Variable_Rate(), input parameter *movie* is a sequence of animation frames, and *transaction_times* is a sequence of times. The function first(*sequence*) used in line 1 returns the first element in a sequence. Function rest(*sequence*) used in line 2 returns a given sequence with its first element removed. The **foreach** loop iterates through the remaining elements of *transaction_times*. Each iteration of the loop displays an animation frame in the sequence. The frame durration is calculated by multiplying the time between transactions by the *dilation_factor* in line 5. Lines 6 and 7 extract a animation frame from the *movie* sequence. Procedure Show() in line 8 is an implementation-specific procedure that displays the *current_frame*. Procedure Wait() halts execution of the algorithm for a time period specified by *frame_duration*. Lines 12 and 13 display the last frame of the *movie* sequence.

3.6 Variable Update Rate and Fixed Playback Rate

At times, it may be desirable to export an animation using a standard fixed frame rate format for insertion into a web page, or for some other purpose. Function Convert(), shown below, converts a variable frame rate animation to a fixed frame rate animation. Basically, the algorithm sees how many times it can chop up each variable length input frame into fixed length output frames. A variable length interval is rounded off if the given output *frame_duration* does not divide evenly. The remainder is saved and added to the next frame time.

In the Convert() function, parameter *frame_duration* is the duration of each frame in the fixed rate output animation. Parameter *dilation_factor* controls the perceived rate of the output animation relative to realtime. The first seven lines of Convert() initialize local variables. In Line 7 the *transaction_times* sequence is artificially extended by one more value. The value is added so the last frame of the output animation sequence will have a duration. The duration of the last input animation frame is arbitrary. In our case, the last input frame is artificially calculated to be equal to the duration of the first input frame. The **foreach** loop iterates through all the remaining transaction times. Each iteration of the loop processes one variable length input frame producing zero or more fixed length output frames. It is possible that a frame may be droped if the duration of an input frame is less than the duration of an output frame. For simplicity, this case is assumed rare and is not considered here. Line 10 computes the dialated duration of an input frame. In line 11, the variable duration input frame is chopped up into fixed duration output frames. Function floor(x) in line 11 returns the integral portion of a decimal number x. In lines 12 through 17, the duration of the input frame is rounded off to a multiple of the output frame duration given by *frame_duration*. The remainder is saved to be added back in durring the next iteration of the loop. The modulo funtion used in line 12 is as defined $\mathsf{mod}(x, y) \equiv x - \mathsf{floor}(x \div y)$. Lines 18 through 21 generate the output animation frames, and lines 22 through 24 move on to the next frame of the input animation. The resulting output animation is the return value of Convert(). The output animation can be played using procedure Play_Movie().

Figure 5 shows a trace of function Convert() on some example input. To see how this works, consider the example from Section 3.1. In the example the first update transaction occurs after one minute and the second update occurs after another three minutes. If time is measured in milliseconds, then the input parameter *transaction_times* is the sequence (0, 60000, 240000). Let input paramter $frame\_duration = 62.5ms$ (16 frames per second), and parameter $dilation\_factor = 0.01$. Parameter $dilation\_factor = 0.01$ corresponds to a speed increase factor of 100. The left column of Figure 5 shows at what line number the action for that row was performed. A number in a cell indicates a variable set to a new value. Boolean values indicate an expression evaluation result. Variables not affecting the control flow are not shown.

```
  function Convert($movie, transaction\_times, frame\_duration, dilation\_factor$)
    : return movie sequence
    begin
1.    $movie\_out \leftarrow$ NULL
2.    $remaining\_delta \leftarrow 0$
3.    $current\_frame \leftarrow$ first($movie$)
4.    $movie \leftarrow$ rest($movie$)
```

```
5.  $current\_time \leftarrow \mathsf{first}(transaction\_times)$
6.  $transaction\_times \leftarrow \mathsf{rest}(transaction\_times)$
7.  $transaction\_times \leftarrow transaction\_times \mid (\mathsf{last}(transaction\_times)+$
                                   $\mathsf{first}(transaction\_times) - current\_time)$
8.  foreach $next\_time$ in $transaction\_times$ do
9.    begin
10.     $delta \leftarrow ((next\_time - current\_time) * dilation\_factor)+$
                                                      $remaining\_delta$
11.     $frame\_count \leftarrow \mathsf{floor}(delta \div frame\_duration)$
12.     $remaining\_delta \leftarrow \mathsf{mod}(delta, frame) * frame\_duration$
13.     if $remaining\_delta > frame\_duration \div 2$ then
14.       begin
15.         $frame\_count \leftarrow frame\_count + 1$
16.         $remaining\_delta \leftarrow remaining\_delta - frame\_duration$
17.       end
18.     for $i \leftarrow 1$ to $frame\_count$ do
19.       begin
20.         $movie\_out \leftarrow movie\_out \mid current\_frame$
21.       end
22.     $current\_frame \leftarrow \mathsf{first}(movie)$
23      $movie \leftarrow \mathsf{rest}(movie)$
24.     $current\_time \leftarrow next\_time$
25.   end
26. return $movie\_out$
  end
```

4 Conclusion

Procedure Process_Variable_Rate() is used to create a variable frame rate animation. Procedure Play_Variable_Movie() is used to play that movie. These two algorithms have an advantage in the Spatial Spreadsheet over other digital animation methods that use bitmaps or fixed frame rates. In the Spatial Spreadsheet, the frames are maintained as display layers. These layers are stored in an in-memory tree data structure that allows for zooming and panning. For each frame, the data structure is quickly traversed and the features valid for a given time are displayed. In this way, the user can play an animation of query results in the Spatial Spreadsheet, then stop the movie, pan and zoom, and then replay the movie from the new perspective without loss of fidelity. This allows for a more interactive animation. If the frames were mere bitmaps, then the image would become grainy when zooming in too close. A variable frame rate movie may be converted for export to a fixed frame rate format using the Convert() function described above. A minor difference between variable frame rate and fixed frame rate is the loss of timeing acuracy between frames in a fixed rate format.

The algorithms presented here require materialized views to be recomputed from scratch after each update to the view's base relations. This is acceptable if a sufficiently large percentage of the base relation tuples are altered during a transaction. In cases where only a few tuples in a base relation are changed, it is more efficient to use incremental view maintenance algorithms [4] to update materialized views. These algorithms calculate results using only the data that changed in the base relations to avoid recomputing the entire result from scratch. To accomplish this, many incremental view maintenance algorithms use differential tables. A differential table is a relation associated with a base relation used in the definition of a materialized view. A differential table contains all the tuples deleted or inserted into a base relation during the last transaction. Future

`line number`	$next\_time$	$current\_time$	$delta$	$remaining\_delta$	$frame\_count$	$remaining\_delta > frame\_duration \div 2$
2				0		
5		0				
8	60000					
10			600			
11					9	
12				37.5		
13						true
15					10	
16				-25		
24		60000				
8	240000					
10			1775			
11					28	
12				25		
13						false
24		240000				
8	300000					
10			625			
11					10	
12				0		
13						false
24		300000				

Figure 5: Example trace of procedure Convert()

work includes the development of movie mode algorithms to take advantage of incremental view maintenance differential tables to improve efficiency.

References

[1] W. G. Aref and H. Samet. Efficient window block retrieval in quadtree-based spatial databases. *GeoInformatica*, 1(1):59–91, April 1997.

[2] O. Buneman and E. Clemons. Efficiently monitoring relational databases. *ACM Transactions on Database Systems*, 4(3):368–382, September 1979.

[3] C. S. Campbell and S. L. Egbert. Animated cartography: Thirty years of scratching the surface. *Cartographica*, 27(2):24–46, 1990.

[4] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD Conference*, Washington, D.C., May 1993.

[5] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proceedings of the ACM SIGMOD Conference*, pages 237–248, Seattle, WA, June 1998.

[6] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999.

[7] G. Iwerks and H. Samet. The spatial spreadsheet. In *Visual Information and information Systems: Third International Conference Proceedings, VISUAL'99*, pages 317–324, Amsterdam, The Netherlands, June 1999. Springer-Verlag.

[8] B. Jobard and W. Lefer. The motion map: Efficient computation of steady flow animations. In *Proceedings of Visualization '97*, pages 323–328. IEEE, October 1997.

[9] M. Kang and S. Servign. Animated cartography for urban soundscape information. In *Proceedings of the 7th Symposium on Geographic Information Systems*, pages 116–121, Kansas City, MO, November 1999. ACM.

[10] A. Koussoulakou and M. J. Kraak. Spatio-temporal maps and cartographic communication. *The Cartographic Journal*, 29:101–108, 1992.

[11] M. Kraak and A. M. MacEachren. Visualization of the temporal component of spatial data. In *Proceedings of SDH 1994*, pages 391–409, 1994.

[12] K. Ma, D. Smith, M. Shih, and H. Shen. Efficient encoding and rendering of time-varying volumn data. Technical Report NASA/CR-1998-208424 ICASE Report No. 98-22, National Aeronautics and Space Administration, Langley Research Center, Hampton. VA, June 1998.

[13] A. M. MacEachren, F. P. Boscoe, D. Haug, and L. W. Pickle. Geographic visualization: Designing manipulable maps for exploring temporally varying georeferenced statistics. In *IEEE Symposium on Information Visualization, 1998, Proceedings*, pages 87–94,156. IEEE, 1998.

[14] A. M. MacEachren and D. DiBiase. Animated maps of aggregate data: Conceptual and pratical problems. *Cartography and Geographic Information Systems*, 18(4):221–229, 1991.

[15] R. E. Meisner, M. Bittner, and S.W. Dech. Visualization of satellite derived time-series datasets using computer graphics and computer animation. In *1997 IEEE International Geoscience and Remote Sensing, 1997. IGARSS '97. Remote Sensing - A Scientific Vision for Sustainable Development*, pages 1495–1498, Oberpfaffenhofen, Germany, August 1997. IEEE.

[16] R. E. Meisner, M. Bittner, and S.W. Dech. Computer animation of remote sensing-based time series data sets. In *IEEE Transactions on Geoscience and Remote Sensing*, pages 1100–1106, Oberpfaffenhofen, Germany, March 1999. IEEE.

[17] F. Schroder. Visualizing meteorological data for a lay audience. *IEEE Computer Graphics and Applications*, 13(2):12–14, September 1993.

[18] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw-Hill, New York, third edition, 1996.