A drag-drop-formula tool for custom visualization

Soren Lauesen, Mohammad Kuhail, Kostas Pantazos, Shangjin Xu, Mads B. Andersen
The IT-University of Copenhagen
{slauesen, moak, kopa, xush, mban}@itu.dk

Keywords: Data visualization, database, interaction, user interface, end-user development.

Abstract: Popular tools for constructing user screens use the drag-drop-set-property principle. The developer drops components (buttons, text boxes, etc.) on the screen and defines their properties, e.g. position, color and text. Then the screen looks right, but it has little functionality. If you want real functionality or a custom-made visualization, you have to switch to tools that are more like programming. Many domain experts are familiar with the popular tools and would like to provide functionality and visualize data, but they are uncomfortable with programming. Could we improve the drag-drop-set-property principle and cover their needs without asking them to program? This paper presents a tool (uVis) that takes a long step in this direction. The principle is to allow each property to be a formula that computes position, color, etc. A formula can combine data from several database tables with data about components and data entered by the end-user. Although aimed at non-programmers, the tool can also boost the performance of programmers. The paper presents the formula language, the tool and some user evaluations of the tool.

1 INTRODUCTION

A visualization consists of basic components (lines, curves, boxes, labels, wedges, etc.) each with its own property values (position, size, color, etc.). Components may be 2D, 3D or 4D (animation). The available components vary from tool to tool, but this is not important for our discussion in this paper.

In order to create a specific visualization, we must somehow put the necessary basic components on the screen and define their properties. *Programmatic* tools require that the developer writes a kind of program that creates the components and sets their property values. *Drag-drop-set-property* tools allow the developer to manually drag and drop the components and set their properties. *Predefined visualizations* contain a program that creates a pattern of components and lets the developer specify some of the properties. Combinations of these principles exist too, of course.

Our focus in this paper is *custom visualizations* where predefined visualizations cannot support the end-users adequately. Figure 1 shows two examples from the medical area. The Lifeline screen shows the patient's notes, diagnoses and medicine on a time scale with multiple, dragable zoom areas. Icon shapes and color indicate the kind of note, the height of boxes indicate the medicine dose, etc. You can see at a glance which medicines the patient gets now, how long the patient has got them, and how they time-wise relate to diagnoses and notes. The data exist already in the database; we just show them in a different way. As far as we know, no health record system uses such screens today.

The bronchial screen shows where biopsies have been taken, how they were taken, and what the lab results tell. Clinicians use it also to record the biopsies and lab results. The diagram of the bronchia is a simple drawing made in the department.

From a theoretical perspective these visualizations are not novel, just variations of known themes such as Lifelines (Plaisant et. al., 1998) and geographical maps. Yet they are highly useful in their domain, and at present they can only be made with programmatic tools.

Ideally we would like a tool where end-users could make such visualizations. However, it is not realistic that any end-user could make visualizations. Our goal is that local domain experts with some IT expertise could make these visualizations in cooperation with local end-users. We will use the term *local developers* to mean these local experts. They are familiar with popular tools such as MS Access and Excel, and they sometimes make small applica-

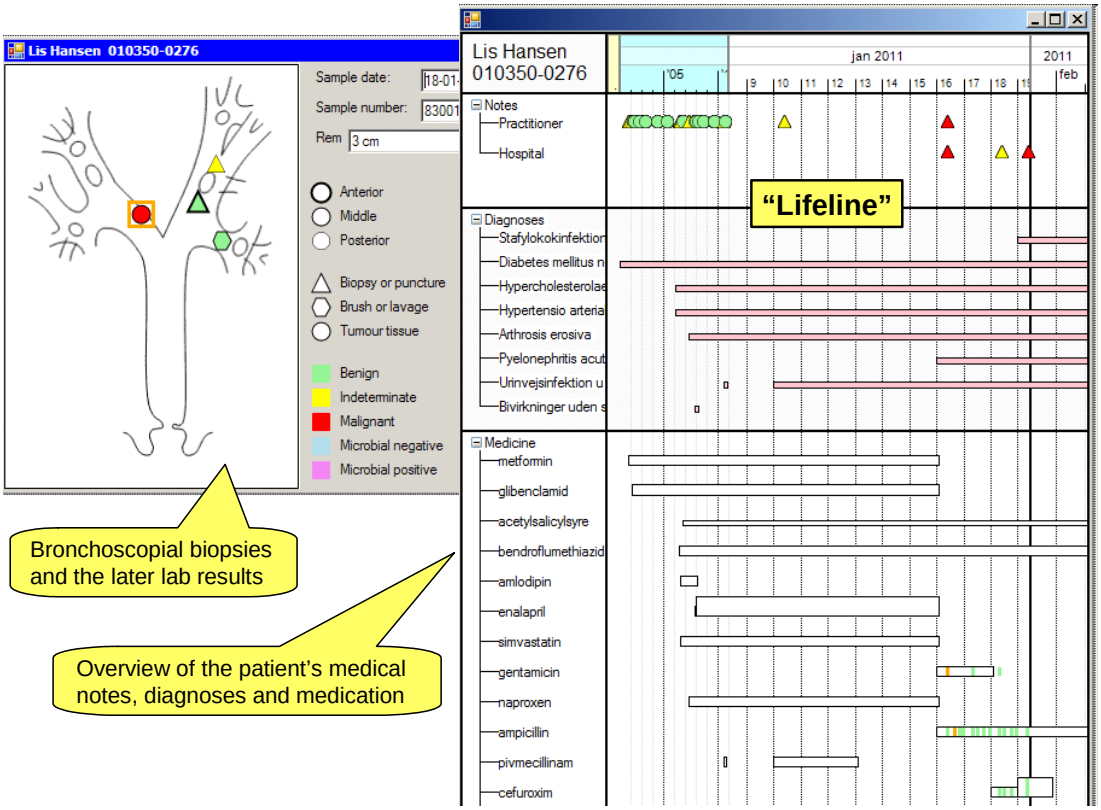

Figure 1. Two screens with custom visualization of medical data

tions for their own use or for use in the department. They are rarely comfortable with programming and with present tools they cannot make custom visualizations such as Figure 1.

Myers et al. (2000) gave an excellent overview of user interface tools in 2000 and explained why drag-drop-set-property tools (called *interface builders* and *interactive graphical tools*) were much more successful with local developers than program-based tools. They also report that spreadsheets are the only kind of "programming" widely accepted by end-users. In this paper we do not consider spreadsheet formulas real programming.

What is the situation today? A recent study (Confidential, 2012a) showed that local developers (called *savvy users*) still need better tools and more attention from the information visualization community.

Why are drag-drop-set-property tools so popular? Most people give up programming. They cannot see the connection between the program and the visible result, and they cannot write a program themselves. Norman describes this as the gulf of evaluation and the gulf of execution (Norman, 1988).

Uvis is a drag-drop-set-property tool where the developer can specify a formula for each property. This formula computes and sets the value of the property. He is also able to specify formulas that create a bundle of components. A formula corresponds to a spreadsheet formula and uses a similar notation, but it is able to combine data from databases, visual components and end-user input.

Using uVis we developed a fully functional version of the two screens in Figure 1 in 4+6 hours. The database existed already. With a bit of training, local developers in the hospital could have made it. Only formulas were needed, no real programming.

2 RELATED WORK

Industry tools such as Microsoft Visual Studio, Eclipse and NetBeans allow developers to construct user screens with drag-and-drop of text boxes, buttons and other components. For each component the developer sets size, position, color and other properties to a constant. This approach can quickly generate a mockup that looks right, but shows only dummy data. In simple cases you can connect a database table to a component, for instance to make a combo box or a data grid component. However, to

make something like the Lifeline screen, "programming behind" is needed and only professional programmers can do this.

Standard graphical presentations such as pie charts and bar charts are provided in Excel, Google Spreadsheets (Google Visualization API, 2012) and as separate packages. These tools don't require programming skills and are widely used. However, to integrate them with a production application, the end-user has to copy and paste data from the application to the tool (or a programmer has to write code that does it). Without programming there is no way to create visualizations beyond what is predefined. For instance, something like the Lifeline couldn't be made. Further the end-user has little interaction with the data and cannot feed data back to the existing application.

Data analysis tools such as Tableau (2011), Polaris (Stolte et al.,2008), Spotfire (2011) and Omniscope (2011) integrate well with existing data and help users explore the data. They don't require programming skills. Here too there is no way to create visualizations beyond what is predefined and something like the Lifeline couldn't be made. Further there is only predefined interaction with the data and the end-user cannot feed data back to the existing application.

Graphics libraries such as GDI+ and Java 2D are available for many programming languages. They provide basic components such as line, polygon, and ellipse. By means of a program you can make them create any visualization, bind to any data and perform any interaction. However, to accomplish this, you must be a professional programmer.

Visualization Toolkits allow you to construct traditional and new visualizations in a programmatic way that is simpler than using graphics libraries, for instance by means of a domain-specific programming language. Examples are Protovis (Bostock and Heer, 2009), D3 (Bostock et al., 2011), Prefuse (Heer et al., 2005), Improvise (Weaver, 2004) and Infovis (Fekete, 2004). These toolkits don't use a drag-and-drop approach, they are not easy to integrate with existing relational data, and you cannot feed data back to the existing application without programming.

3 UVIS

In this section we will explain how uVis works from the developer's point of view and the design rationale behind. Figure 2 shows uVis Studio when the local developer has constructed the trivial part of the bronchoscopy screen: the components that occur only once. He has for instance dropped an icon component in the left part of the screen and set its File property to show the bronchial diagram. This is quite similar to how popular tools work today.

The IT department has provided a *data map* file that has a connection string to the database and specifies the available tables and relationships. The local developer will rarely have the skills and rights to make a data map file.

During construction the local developer sees the bronchoscopy screen exactly as it will look to the end-user. He can also interact with it as the end-user would do. Uvis uses the data map file to show a map of the tables in the database and how they relate to each other.

The developer's next task is to make the system create an icon for each biopsy. The number of icons is dynamic; it depends on the database contents.

3.1 Dynamic component creation

How can we dynamically create components? Since we want to use spreadsheet principles, we may ask how components are created in a spreadsheet. The answer is easy: they are not created - they exist from the beginning. A "component" in a spreadsheet is a cell, and all the cells exist conceptually from the beginning. You can make new tab sheets in a spreadsheet, but this is done manually (unless you are able to write a macro - a piece of program - that does it). In spreadsheets you don't have loops. Instead the user copies cells the necessary number of times.

Visualization toolkits create visual components dynamically. They use a set of rows or an array of data to create a corresponding set of components. The crucial point is how the rows are generated. Current tools use a programmatic way, which is a cognitive barrier to local developers. We want to specify the rows as a formula in a component property.

Figure 2 shows how the local developer creates dynamic components with uVis. He has dragged a Glyph component from the toolbox and dropped it on the bronchial diagram. A Glyph can appear as different shapes: circle, triangle, etc. Initially it appears as a gray hexagon. The property grid shows the properties of the Glyph. The developer has set the name of the component to *sample*. Now he sets a special repeater property, Rows, in this way:

```
Rows: Bronchial Where ptID = Param[0]
```

Bronchial is a table in the database that has a row for each biopsy. It has a *ptID* field that identifies the patient. The bronchial screen was opened with the current *ptID* as a parameter, *Param[0]*. The result of

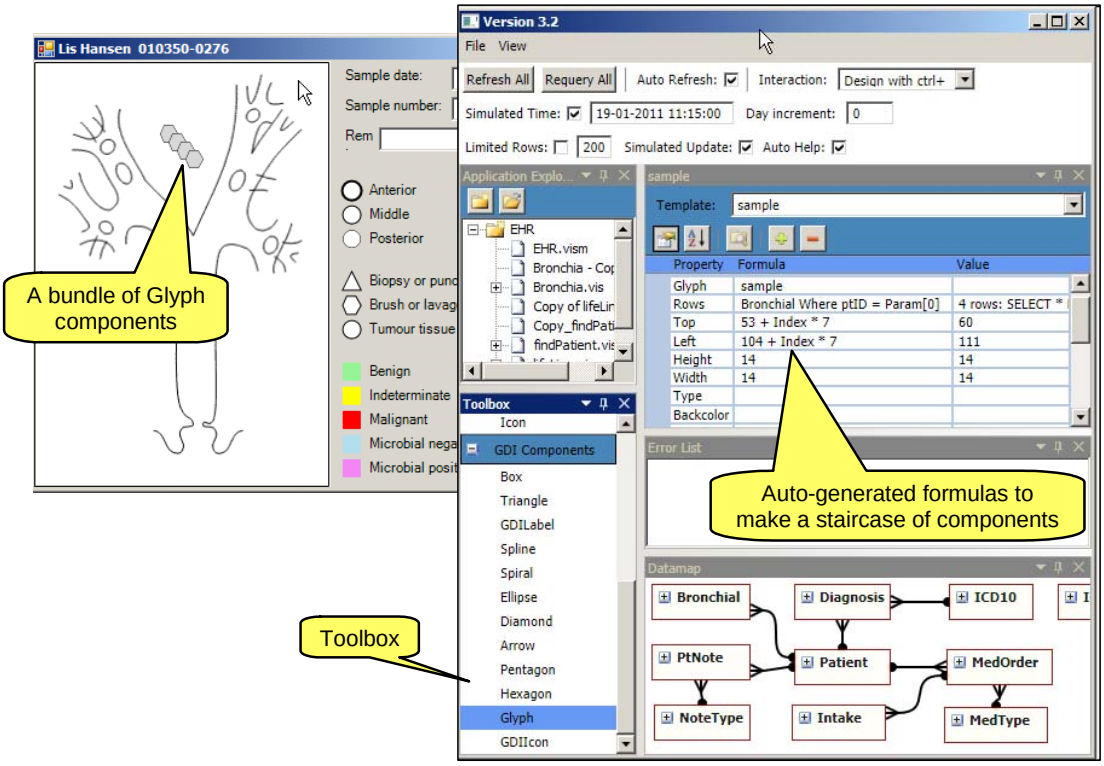

Figure 2. Setting the Rows property to create a bundle of components

the Rows formula is the set of Bronchial rows for the current patient. As soon as the developer had typed the Rows formula, uVis generated a Glyph component for each row and connected it to the row. Furthermore, uVis showed the Glyphs as a staircase of gray icons to visualize how many there are.

If the developer had to express the Rows formula in the usual way as an SQL statement, it would look like this:

```
SELECT Bronchial.ptID, Bronchial.y, Bronchial.x, Bron-
chial.kind, Bronchial.result, Bronchial.ant_post, Bron-
chial.splDate, Bronchial.splNumber, Bronchial.remark
FROM Bronchial WHERE [Bronchial.ptID] = 0103500276
```

The developer could have written SELECT * instead of listing all the fields he needs, but this would retrieve all fields in the database. In a real-life database with many fields, this may be very slow.

He would also have to insert the actual patient ID (0103500276) into the SQL statement in a programmatic way.

Uvis generates the SQL statement automatically based on the formula, collects the necessary fields from all the formulas in the screen, and inserts them as the SELECT part. Although the uVis formula still has the flavor of an SQL statement (the Where clause), it is much simpler to write than the real SQL statement.

The Rows property can specify a set of rows in other ways. Some correspond to database joins, explained later. The simplest Rows property is a number, e.g.

```
Rows: 12
```

It generates 12 components. They could for instance be labels for the months of a year.

When uVis creates components with a Rows formula, they become a *bundle* of components. Each component in the bundle has an *Index* property that is 0 for the first in the bundle, 1 for the next, etc.

When uVis has created the components, it calculates the property formulas and sets the property values. In our example, uVis Studio automatically defined the formulas for Top and Left so that the icons appeared as a staircase:

```
Top: 53 + Index * 7
Left: 104 + Index *7
```

In this way the first component got Top = 53 (Index = 0), the next Top = 60, etc.

Originally uVis Studio didn't make such an automatic staircase, but we observed that developers were puzzled when they defined the Rows property.

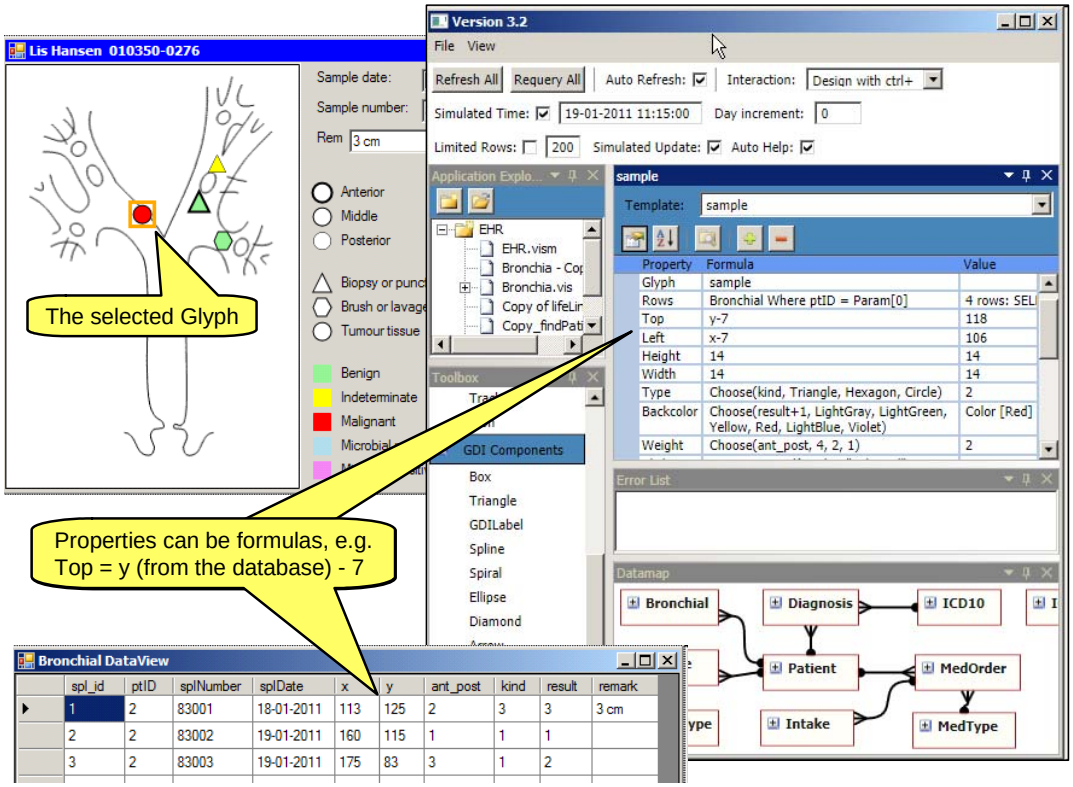

Figure 3. Using database fields to set position, shape and color

Nothing seemed to happen because all the components were on top of each other. The automatic staircase helped.

3.2 Formulas and data addressing

In general, a formula is a function that takes some data as input and computes a result. In the visualization world, we must be able to address these data:

1. External data (e.g. in a database)
2. Property values in the same or other components.
3. Data provided by the end user, e.g. the position of a scroll bar or the contents of a data entry textbox.

Which functions can we express with formulas? In principle any computable function, but in order to reach non-programmers we restrict ourselves to spreadsheet-like formulas. They are quite powerful, but they cannot compute everything. For instance there is no recursion, so some functions cannot be computed. For performance reasons there are also limitations on how we can address databases. We will only address databases in ways that can be translated into efficient queries.

We will illustrate database addressing with the *sample* component. Figure 3 shows uVis Studio when the local developer has created the *sample* components and defined the property formulas. He has selected one of the icons that represent a sample. He sees its property formulas in the property grid and also the actual property value. The formulas are the same for all the components, but the values vary between components in the bundle. He has also opened the bronchial table. The selected icon corresponds to the first row of the table.

The Left position of the icon is computed by the formula *x-7*, where x is a field in the database. For this specific icon, the formula gave 106 as the result and uVis put the icon 106 pixels from the left border of the user screen.

The shape of the Glyph is computed by the Type property. The formula retrieves the *kind* field from the database and chooses the corresponding shape: triangle, hexagon or circle. When the developer has typed the formula or changed it, the system updates the user screen immediately.

A formula may refer to formulas in other components, which again may refer to other formulas. Just as in spreadsheets, circular references may occur. Uvis shows this and other errors in the error list panel and as colored marks in the formulas.

Language style: We have based the formula language on Visual Basic because it is widely known among the developers we aim at. Parts of the Visual Basic language are also used in Excel and some SQL query languages. This choice means that the developer can type names without caring about upper/lower case, and uVis gives feedback by correcting them to the proper case. It also means that "=" means assignment or comparison depending on the context, comments are shown by a single quote ('), and other trivial details.

Most of the functions available in Visual Basic are also available in uVis, for instance Sin( ), ToDay( ), Format( ) and Choose(i, a1, a2 …).

Saving the screens: When the developer closes the bronchoscopy screen or closes uVis Studio, uVis saves the screen as a .vis file that can be read with simple tools, e.g. notepad. Figure 4 shows part of the .vis file for the bronchoscopy screen.

3.3 End-user data and interaction

We will show an example of how the end-user can interact with the screen. This also illustrates how a formula can address properties in other components.

The developer has decided that when the end-user clicks a *sample* icon, it should be marked with an orange frame. The details of the sample should be shown in the text boxes at the top right (Figure 3).

The key part of this is a Glyph component that serves as a Marker. The developer has given it these property formulas:

```
Glyph:       Marker
selected:    Init -1 ' The selected sample
Visible:     selected >= 0
Top:         sample[ selected ].Top-3 Default 0
Left:        sample[ selected ].Left-3 Default 0
Height:      20
Width:       20
Type:        Square
Weight:      3
BorderColor: Orange
BackColor:   Transparent
```

The developer has added his own property, *selected*. It is not a built-in Glyph property such as Top and Type, but what we call a *designer* property. *Init -1* means that *selected* initially is -1, but the value can change as a result of end-user actions. When the end-user selects a *sample* icon, *selected* should become the Index of the icon.

Visible is a built-in property. The formula says that the Marker should be visible when something is selected (*selected >= 0*). Initially it will be invisible.

Top says: Walk to the bundle of samples. Take the icon with the index given by *Selected*. Take its Top property value and subtract 3 pixels to make the orange frame surround the Glyph. If this doesn't work, for instance because nothing has been selected, use the default value and make Top = 0. This is an example of addressing a property in the same component (*selected*) and in another component (*sample[i].Top*). Notice that uVis can address specific items in a bundle as if it was an array.

The remaining properties should now be obvious. The result is that an orange square shows around the icon when it is selected.

When an icon is selected, the screen should also update the text boxes at the top right. The developer handles it with formulas like this one for the *sample-date* text box:

```
TextBox: splDate
…
```

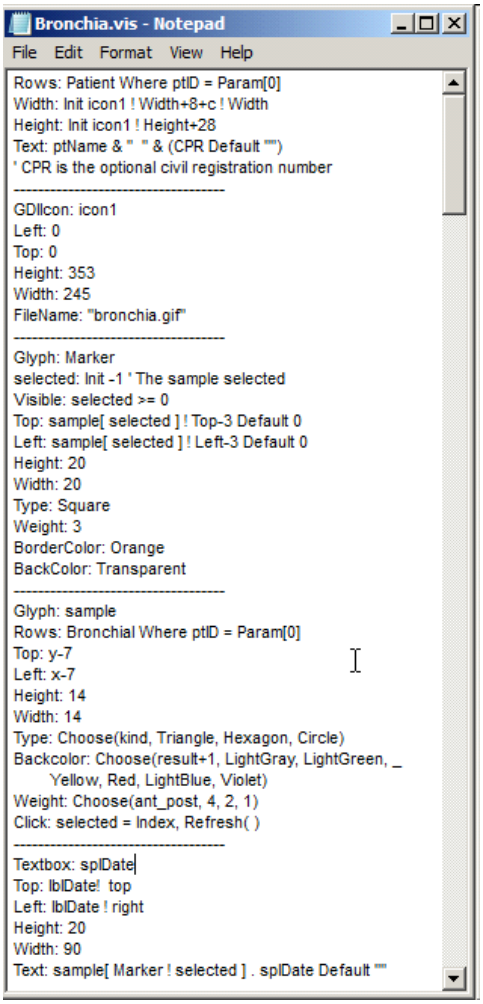

Figure 4. Saved vis-file

```
Text: sample[ Marker.selected ].splDate Default ""
```

The Text property specifies what to show in the text box. The formula says: Walk to the bundle of samples. Also walk to the marker glyph and get its *selected* value. Use it as the Index in the bundle to get the selected icon. Finally walk to the data row connected to the icon and get its sample date (*splDate*).

The walk principle: Data references in uVis formulas use the walk principle: The system walks from object to object to get the result. The Text formula above is an example of this. The formula walks to a bundle of components, then to another component to get the index and use it to select a component in the bundle, and finally to a row connected to the component to get the desired field.

Dot-operators and name ambiguity: Understanding a formula such as the Text formula above requires good knowledge of what are visual components and what are database elements. To help the developer, uVis can change the dots in the formula to show what is what. A dot (.) means database elements and a bang (!) means visual component elements. Using these dot-operators, uVis would show the Text formula like this:

```
sample[ Marker ! selected ] . splDate Default ""
```

With a bit of training, the developer can see at a glance that "selected" is a component property and "splDate" is a database field. The dot-operators also help resolving name ambiguity. Assume that the database had a field called *Top*. The developer cannot change this name, nor can he change the property name *Top*. But he can use bang or dot to tell the compiler whether he means a component property or a database field.

Event handler properties. We only lack one thing to make the selection construction run: a way to set *Selected*. This is done through the *sample* component. It should respond when the end-user clicks it. The developer has defined an event handler property for it:

```
Glyph:   sample
…
Click:   selected = Index, Refresh( )
```

When the end-user clicks a *sample* icon, uVis performs the *statements* in the click formula. As a result, *selected* will become the index of the clicked icon. The statement *Refresh( )* asks uVis to re-compute all formulas and redraw components where a property value has changed. As a result the Marker will appear or move, and the text boxes will show data about the selected sample.

In contrast to ordinary formulas such as *Top*, an event handler formula cannot be evaluated at any time. An event handler is evaluated only in response to an end-user action. In industrial tools, components can send a lot of events to each other, for instance *BorderChanged* and *ValueChanged*. Uvis has no such events because all updates are made with *Refresh( )*, which re-computes and redraws as needed. This is the spreadsheet principle, but it assumes good performance (see section 5).

Event handler statements include setting a value in a property or a database field, opening a form, or committing a database transaction.

Depending on the application, it may be necessary to perform actions beyond the built-in uVis statements, for instance to send an email or transmit data to/from an external system. This requires that someone makes a piece of real program, tests it and exposes it as a method that can be called from an event handler formula. This is "programming behind", but it is used far less than in the traditional approaches.

3.4 Joins - walking among tables

Above we showed examples of walking between visual elements. It also makes sense to walk between database tables.

The data map shown in Figure 3 is an entity-relation diagram (E/R diagram). The tables are connected with crow's feet and it makes sense to walk along these feet. As an example, the Patient table has a one-to-many crow's foot to the MedOrder table. It symbolizes that we can walk from a patient to many medicine orders. Walking the other way from a MedOrder to Patient, we come to only one patient.

Let us look at a tough example, the medicine orders at the bottom of the Lifeline (Figure 1). The developer has connected the Lifeline screen to a single patient with this formula:

```
Rows: Patient Where ptID = Param[0]
```

He now wants to generate a box for each of the patient's medicine orders. He drops a box in the lower part of the Lifeline screen and gives it these properties:

```
Box:    MedOrderBox
Rows:   Parent -< MedOrder
```

The Rows formula means: Start in the patient row connected to the screen (the screen is the Parent in this case). The -< symbolizes a one-to-many crow's foot. Now walk along the crow's foot to the MedOrder table. The result is a set of rows, one for each

of the patient's medicine orders. Each row contains fields about the medicine order (the type of medicine, the start and stop time for the medication, the amount per dose, the dose, and the number of times per day).

The result is a staircase of medicine orders. Next the developer aligns them to the time scale at the top of the Lifeline with these formulas:

```
Left:    timeScale ! HPos(startTime)
Right:   timeScale ! HPos(stopTime)
```

The Left formula means: For this medicine order, walk to the time scale component at the top of the Lifeline. Call its horizontal position property (HPos) and ask it to translate the start time of the medication to a pixel position. Use this position as the Left property. The Right formula works the same way. The result is that each MedOrderBox is stretched correctly in the time dimension.

What about *Width?* The developer can specify any two of Left, Width and Height because we observed that they often tried to do so.

Next the developer wants to show the amount of medicine as the height of the boxes. To do this he has to multiply the amount per unit, the dose and the times per day. These fields are readily available in the medicine order row. However, is this a large or a small dose? To indicate this, he needs access to the MedType table that contains the DDD (normal daily dose). Fortunately the E/R diagram has a crow's foot from MedOrder to MedType. He can just continue walking from MedOrder to MedType.

To do this, he changes the Rows formula and can then include DDD in the Height formula:

```
Box:     MedOrderBox
Rows:    Parent -< MedOrder >- MedType
Height:  (amount * dosage * timesPerDay)
         /DDD * 8
```

The Rows formula now gets the MedOrder rows as before and then walks on for each MedOrder row to the related MedType. It includes the relevant MedType fields in each of the rows, e.g. DDD.

The last step of the Rows formula is a one-to-many relation, symbolized with >-. If the developer makes a mistake and types -< instead, or dot or bang, the compiler will replace it with >- to show what kind of relation it actually is.

In practice, there are often missing data in the tables. As an example, the real medicine data we work with often lack a reference to MedType. The visualization can easily deal with this by means of the *Default* operator explained above.

If the developer had to write the SQL statement for this Rows formula, it would look like this:

```
SELECT … FROM MedOrder LEFT JOIN MedType
ON MedOrder.medID = MedType.medID
WHERE [MedOrder.ptID] =0103500276
```

Some programmers suggest that Microsoft LINQ would be more compact. Maybe, but not in these cases. A LINQ join also has to specify the join condition in the same way as SQL.

3.5 Walking from data row to control

Above we have shown how a uVis formula can walk from a visual component to a database field (with the Rows formula and a dot to refer to the field), walk among visual components (with the component name and an optional index), and walk between table rows (with the join operators -< and >-).

In some cases it is also necessary to walk from a table row to a visual component. The MedOrderBox provides an example. To align the box vertically to the medicine names at the left, the developer writes this formula for the Bottom property:

```
Box:     MedOrderBox
…
Bottom:  MedType -= PatientMedicines ! Bottom
```

The Bottom formula means: For this medicine order, walk to the MedType part of its data row. Then walk to the medicine label (PatientMedicines) that is connected to the same MedType. Finally, use the label's bottom position as the medicine order's bottom.

4 DEVELOPMENT SCENARIO

In this section we show how a local developer would use uVis to construct the screens in Figure 1 and deploy them in the department.

Connecting to the database

First the local developer needs a *data-map* file that gives uVis access to the existing database or to equivalent system services. Most likely he will use an anonymized test version of the database. The central IT department would give him the necessary permissions and help him set up the data-map file.

Next he opens uVis Studio and tells it to select the data-map file. His computer screen will now look like Figure 2, but the bronchial screen is an empty default screen.

Constructing the screens

The local developer will now drag and drop components on the screen and set their properties as explained above. He can develop several screens at the same time.

Testing the screens
The local developer has made the basic test of the screens while he constructed them. Whenever he typed or changed a formula, the system immediately retrieved data from the database and showed it exactly as the end-user would see it.

However, the local developer should make additional tests of the screens. He should make usability tests of the screens to ensure that they are understandable. He should test that the screens show abnormal data correctly; preferably he should have a small test database with abnormal data for this. And he should test that the queries don't overload the database. Here he needs a large test database for measuring performance.

We have paid much attention to the test situations. For instance, it is easy to switch between databases. You make a copy of the data-map file for each database and modify the database connection string. When you run a test, you simply open the proper map file. You can run uVis in various test modes, for instance simulate that the date and time is something that matches the data you test with. You can avoid that uVis makes real database updates, but only simulates that data has been updated. To avoid that a database query by mistake takes a very long time and blocks testing, you can limit database access so that the system never retrieves more than for instance 200 rows from a table.

Let the clinical staff use the screens
After testing, the local developer has to get the necessary rights to access the production database. He also sets up a map file for production. Further, the end-users need rights to open the map file and access the production database.

He puts the map-file and the vis-files in a folder on the department's file server. The local users can now open the map file in the same way as they would open a Word file. As a result, uVis will connect to the test database and open the start screen that the local developer made. From there they can navigate to other uVis screens.

Close integration with existing applications
In the scenario above, end-users opened a map-file to see the existing data through uVis screens. Another approach is that an existing application asks uVis to open the map-file and show the uVis screens. This requires a small change to the existing application so that it calls uVis when the user for instance clicks a certain button on an existing screen.

Since uVis can boost the work of professional developers, they might use it themselves in existing applications.

5 EVALUATION

We have developed many small applications in an experimental way with uVis and also developed customizable versions of bar charts, pie slices and other traditional visualizations (see confidential 2012c). Most of them became surprisingly simple as we learned to utilize the power of the formula language.

Performance
When the end-user has done something, the event handler will usually ask uVis to refresh everything in the same way as a spreadsheet recalculates all cells. There are various ways to optimize refreshing, for instance only recompute properties that depend on the item changed. At present we don't try to optimize. We get adequate performance with a simple algorithm:

Recompute all formulas, requery the database if an SQL statement has changed, set all component properties to the new computed value (whether it has changed or not), and update the screen accordingly.

The table below shows the performance for the Lifeline with the screen shown in Figure 1 (average of 10 measurements on an ordinary 2GHz PC with 2 GB memory and a local MS Access database). The total time to open the screen is 0.7 seconds including 0.4 seconds to make 8 queries to the database. The time to refresh the entire screen is 0.08 seconds.

Usability for developers
Ease of learning is an issue when we aim at non-programmers. However, usability is not only a property of the formula language. Computerized and human assistance is also very important to help developers get started.

For the last year we have run usability tests of the tool and gradually improved it. We have mentioned how the staircase presentation of component bundles improved understanding, but have improved

Time to open Lifeline, Figure 1	ms
Scan the .vis-file (5500 chars)	23
Compile 180 formulas	20
Compute and create 146 components	133
SQL queries (8 queries, 140 rows total)	401
Show 146 components	94
Total time to open	671

Time to refresh Lifeline	ms
Compute and create 146 components	46
Show 146 components	32
Total time to refresh	78

many other things too, e.g. terminology (*Rows* or something else?), auto-completion where the tool suggests what to type, direct inspection of the database contents and direct inspection of the property values. We have also developed tutorials and other support information.

We have made usability tests with a total of 24 typical local developers (non-programmers) and 6 developers that had experience with other visualization tools. We tested with one user at a time. The tests took between 2 and 3 hours.

In spite of testing with gradually improved uVis versions, we can make some conclusions. All subjects except one could make simple visualizations with uVis, but sometimes had to ask for help. As the complexity increased, fewer could make it within the 2-3 hours. The subjects understood the concepts and could reason about the formulas. However, the join operators (-< and >-) were not as intuitive as we had hoped. This was related to the subjects having a weak understanding of databases in general.

Some subjects said that they would like to experiment more with the tool on their own.

The subjects who had programmed visualizations professionally said that a mature uVis would have saved 90% of their time.

Details of the evaluation are published in (Confidential, 2012b).

Future

Uvis is currently an operational prototype that proves the concept: It is possible to make such a tool, it can become sufficiently easy to use for the target developers, and it performs well on the computer. However, many things are missing before local developers can use it on their own, and the tool is too immature to be taken up by others.

One important limitation is that it at present only runs on the Windows Forms platform.

5 CONCLUSION

We have presented a tool that allows local developers to implement fully functional custom visualizations and end-user interactions without writing program code. During development the tool shows the final screens all the time and allows the developer to interact with the screens in the same way as the end-user. Without changing context, the developer can drag and drop components and set their properties to formulas. A formula can access databases, visual components and end-user input. Access from a formula to program code written for the specific application is possible, but rarely necessary.

REFERENCES

Bostock, M., Heer, J., 2009. *Protovis: A graphical toolkit for visualization.* IEEE Trans. Vis. and Comp. Graphics, 15(6):1121–1128.

Bostock, M., Ogievetsky, V., Heer, J., 2011. *D³ Data-Driven Documents*, Visualization and Computer Graphics, IEEE Transactions on, vol.17, no.12, pp.2301-2309.

Confidential authors, 2012a.

Confidential authors, 2012b.

Fekete, J.-D., 2004. *The InfoVis Toolkit.* Proc. IEEE InfoVis, pages 167–174.

Google Visualization API, 2012. http://code.google.com/-apis/visualization/documentation/gallery.html

Heer,, J., Card, S. K., Landay, J. A. 2005. *Prefuse: a toolkit for interactive information visualization.* Proc. ACM CHI, pages 421–430, 2005.

Myers, B., Hudson, S. E., Pausch, R., 2000. *Past, present and future of user interface software tools.* ACM Transaction on Computer-Human Interaction, Vol. 7, No. 1, pp. 3-28.

Norman, D., 1988. *The psychology of everyday things*, Basic Books, New York.

Omniscope | Visokio, 2011. http://www.visokio.com/omniscope.

Plaisant, C., Heller, D., Li, J., Shneiderman, B., Mushlin, R., Karat, J., 1998. *Visualizing medical records with lifelines.* CHI 98 conference on Human factors in computing systems, CHI '98, pages 28–29, New York, NY, USA.

Tableau, 2011. http://www.tableausoftware.com/

Spotfire, 2011. http://spotfire.tibco.com/

Stolte, C., Tang, D., Hanrahan, P., 2008. *Polaris: a system for query, analysis, and visualization of multidimensional databases.* Commun. ACM 51, 11 (November 2008), 75-84.

Viegas, F. B., Wattenberg, M., van Ham, F., Kriss, J., McKeon, M., 2007. *ManyEyes: a Site for Visualization at Internet Scale.* Visualization and Computer Graphics, IEEE Transactions on, vol.13, no.6, pp.1121-1128, Nov.-Dec.

Weaver, C. E., 2004. *Building Highly-Coordinated Visualizations in Improvise.* INFOVIS 2004. IEEE Symposium on Information Visualization.