Denim-Gabbeh-Multi-users Report

Amir M Naghsh – Gilles Bailly
Sheffield Hallam Univeristy- Grenoble University

Table of content

I Introduction

The main aim of this work was to introduce synchronous collaboration into an electronic paper prototyping tool such as Gabbeh. In other words enabling Gabbeh to support multi entries thorough various events and enabling Gabbeh to support multi-users to use different tools to create, delete, move, resize and edit the design components at the same time. Augmented table and TCLVision technology have been used to provide a collaborative environment (frame work).

It is generally accepted that user involvement is essential for a successful design of an interactive system. Bodker and Gronbeak (1991) found strong arguments for a more active and direct user involvement in designing computer systems. They found that it is important for users to find out how computer systems work. But it is more important that users learn through experiences with such not just to read a system specification or watching a demonstration.
Prototyping is one of the known ways for encouraging users to become more involve in the design process of a computer system. To encourage a better user involvement it is important that prototyping approach supports cooperative activity between users and designers, a possible example is the approach introduced by Bodker et al. (1991), called cooperative prototyping. Such prototyping approach could establish a design process where both users and designers are participating actively and creatively, drawing on their different qualifications. To facilitate such a process, the designers must somehow let the users experience a fluent work-like situation with a future computer application; that is, users' current skills must be brought into contact with new technological possibilities.

Button et al. (1996) explains that a growing number of ethnographic reports suggest that a collaborative design process depends on communications, and on transformation process involving design prototypes. The communication dimension and the role and transformation of artefacts in design work intersect in that the prototypes are subject to discussion, negotiation, and alteration. Perry et al. (1998) explain that design work should proceeds a situation in which joint, coordinated learning and work practice evolve, and in which prototypes help to meditate and organise communication. It becomes an important area in CSCW to provide affordable ways and tools to communicate and collaborate during design process of developing prototypes, mock ups, and other objects and models. Such communication and collaboration depends on team layout (co-located or distributed) and the synchrony of communication.
In a collaborative design process various tools are used to develop mock ups, models and prototypes. Some of these tools are explained in previous studies (Lin et al. 2000) as pens, whiteboards, papers and tables. They have been recognized as primary tools that were used for explaining, developing and communicating ideas during the early stages of the design process.

In previous researches pen based interaction technology was applied in interactive system design which also could be described as "electronic paper prototyping" tools. Examples include Denim, Freeform, Silk and Satin. Further study (Naghsh et al. 2005) has showed how supporting annotation in a distributed electronic paper prototyping environment could support asynchronous communication and encourage more user participation (e.g. Gabbeh).

II Related work

1 Satin

Satin (Landay et al.) is a java based application that was developed at Berkeley University. It is a toolkit support for informal ink applications. Satin was motivated mainly to apply natural activities such as sketching and writing in electronic prototyping environment. Satin could be broken down in to 12 concepts.

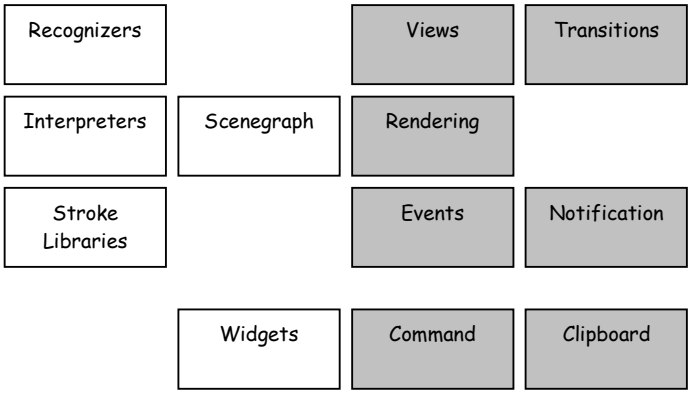


Figure 1: Lecture notes: University of California [1]

Satin assembles stroke from mouse or a pen movement and provides reusable mechanisms for handling and processing strokes. Satin process strokes in following order: first Satin processes stroke with its gesture interpreter if it does not match any of recognizers then it would re-dispatch the stroke to one of its children (the component which contains the stroke), then Satin processes the stroke with gesture interpreter and if it is not recognised as gesture then it processes the stroke with its ink interpreter and handle the stroke in the object itself.

2 Denim

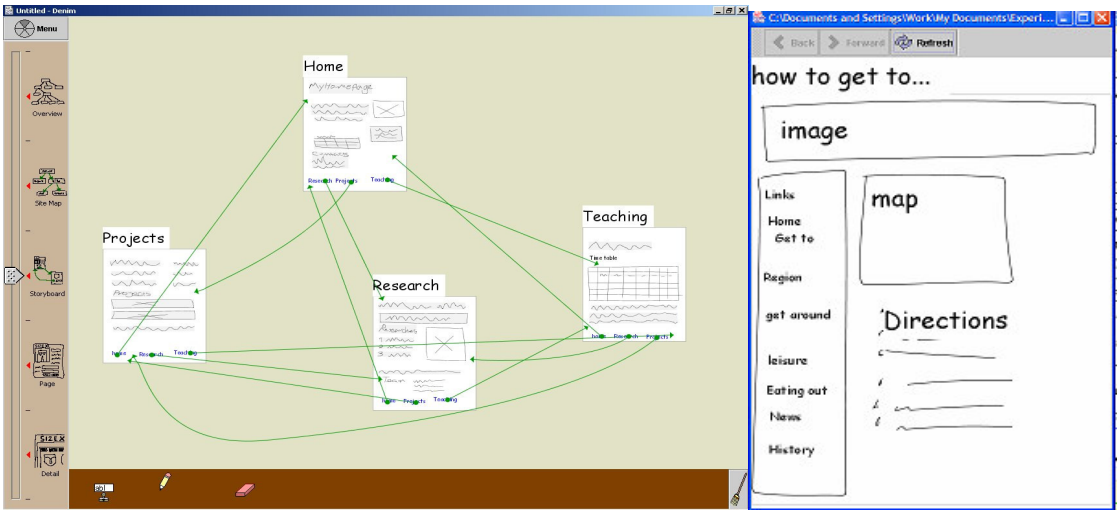


Figure 2 : Denim, run mode, design mode

[1] http://dub.washington.edu/projects/satin/docs/presentations/satin-uist2000_files/v3_document.htm

DENIM [(Landay et al.) is a sketching tool for designing web-sites that has been developed in java. DENIM is usually run on a graphics tablet, such as a TabletPC or a Wacom Cintiq. In DENIM users can sketch out the overall structure of a site (a collection of pages), sketch the contents of the pages as a set of 'scribbles', define hyperlinks from scribbles in one page to another page, and then execute the resulting hypertext in a reduced functionality browser. Figure 2 depicts a screenshot from DENIM. The slider bar to the left of the screen allows the site to be viewed at different levels of detail – varying from a site overview that simply identifies the pages included, through a navigation view where the overall navigation can be examined, down to a detailed view where fine details of individual pages can be manipulated.

3 Gabbeh

Gabbeh (Naghsh et al. 2004) is an extension to DENIM. The core innovation in Gabbeh is that it allows different stakeholders to add arbitrary annotations in the form of comments either when the model is being designed or when the model is being executed. Figure 3 shows an example of comments in Gabbeh in the 'design view'.

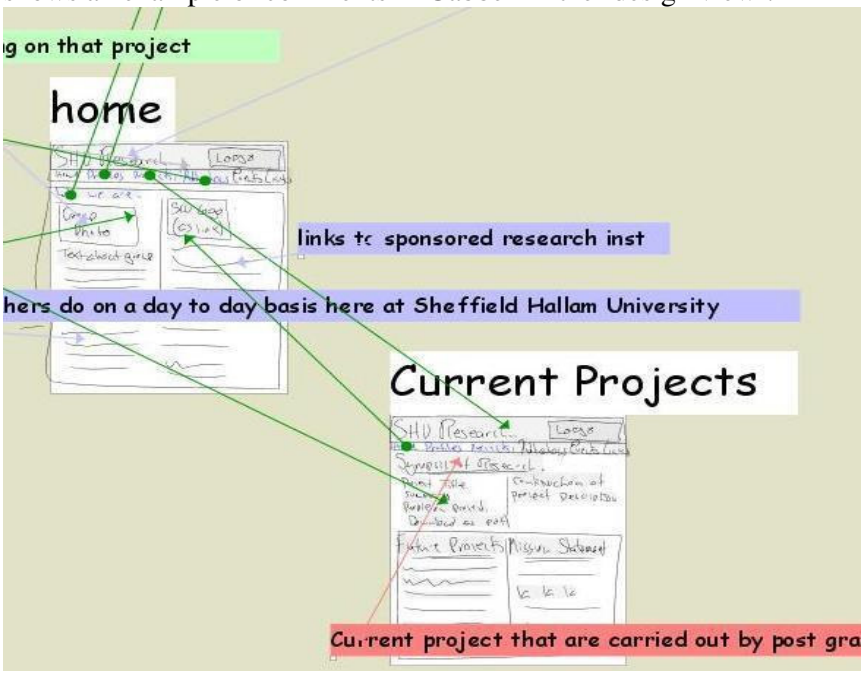


Figure 3 : Comments in the Gabbeh design view

End users may execute Gabbeh using a separate limited functionality browser to review a design. To make Gabbeh easier to be used by end-users, the design sheet is excluded from the version of 'run mode', which is installed at end-users site. It allows the end-users to work only with a simple browser with annotation features

A comment in Gabbeh can be associated with any arbitrary number of design components, such as panels, labels, texts and scribbles. Comments are given a background colour. This is intended to allow development teams to distinguish between different types of comments, or perhaps between comments from different speakers. The usage of comment is deliberately left open.

Gabbeh allows end-users to view and add comments while they are reviewing the design in 'run mode'. This functionality is intended to allow end-users to give feedback through the prototyping medium. Figure 4 shows an example of how users can view and add comment in 'run mode'. Comments are displayed in a side window adjacent to the page. Gabbeh displays

the comments location within the page using coloured numbers on the page. If the comments is only associated with the page, Gabbeh only displays the comment in the side window.

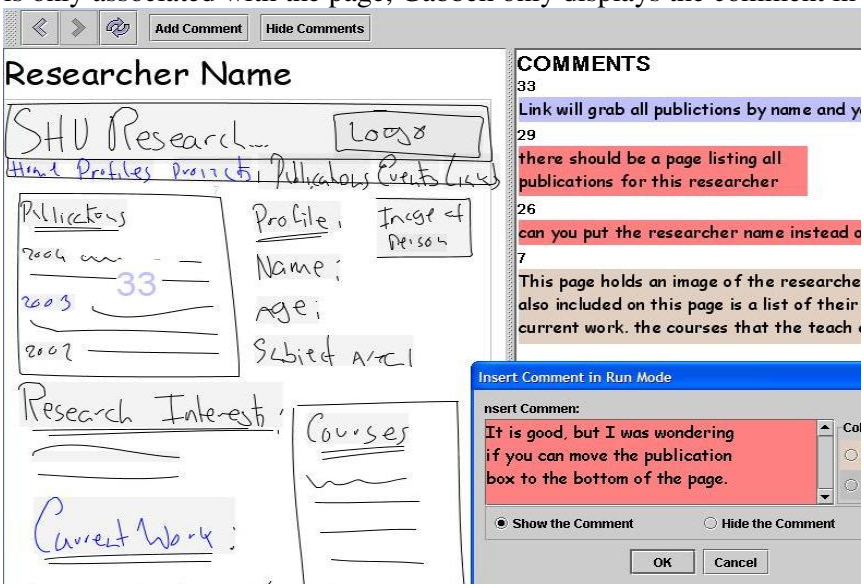


Figure 4 : Adding comment when Gabbeh is executed

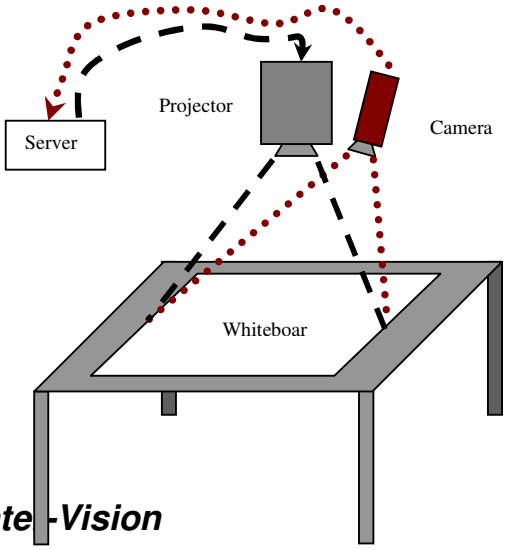
III Implementing Multi-pointer access for Gabbeh

The current version of Gabbeh supports asynchronous communication by letting one individual user at the time to make one entry (annotations) through a mouse, a keyboard or a graphic tablet. As it was explained in the introduction the main aim of this work is to enable Gabbeh to support synchronous collaboration in the early stages of design process. Therefore it is necessary to enable Gabbeh to support multi-entries for multi-users at a same time. One possible way to support multi-entries is to use TCLVision technology which was already available in CLIPS laboratory.

1 TCLVision

TCLVision is a toolbox that is able to track tokens on an augmented table. The augmented table is consisted of a white board, a retro projector and camera. The white board is placed on the top of the table (on the sheet). The retro projector is placed over the desk (fitted to the ceiling) and projects the interface on to the table. The camera is fitted next to the projector and is used to track tokens movement on the table and create event and send them to a socket.

Figure 5: Augmented layout.



2 Multi-Pointe -Vision

Multi-Pointer[2] is a java toolbox which is developed to capture generated tokens by TCLVision and generates *tokenEvent* for any java applications. The tokenEvent indicates that a token action occurred in a component. A token action is considered to occur in a component if the token is over the visible part of the component's bounds when the action happens. This tokenEvent is generated by Multi-Pointer toolbox for three action performed by a token. These three actions can be explained as following procedures:

first action is token *appeared,* for this action, firstly user needs to decide a location on the white board as the position for the action that he wants to perform and then, he should hide the token with his hand (put his hand over the token) completely so camera would not be able to see it while he is moving it around on the table. Then he can move the token around the table until he reaches his interested position. Then user can take his hand away let camera see the position of the token and that's when the token *appeared* action occurs. This action is called appeared because when user uncovers the token, it becomes appear to the camera and the action occurs.

Next action is when token is *moved* on the table. In this action user uses his finger to move the token on the table. It is important that users don't cover the token, and maintain the visibility of the token by camera and let the camera track the token during the move.

The last action is token *disappear.* This is when users have managed to perform his goal (e.g. drawing a line) and wants to conclude it. He has to hide the token again by covering it. When user covers the token with his hand, token *disappears* when camera can not see it any more and the TCLVision recognizes the action since camera doesn't receive any more signal for that particular token.

For each of these actions, a token is *appeared*, a token is *disappeared* and a token is *moved* on the augmented table, a *tokenEvent* is passed to every *TokenListener* object which is registered to receive the token events. Each listener object gets a *tokenEvent* containing the token event. The *tokenEvent* has three attributes which are *tokenID*, *tokenPosition* and *tokenState*. The

[2] In this document *Multi-Pointer* is also used to refer to *Multi-Pointer-Vision* package.

tokenState indicates the performed action which is one of the following: *Appear*, *Disappear* and *Motion*.

Multi-Pointer is consisted of four classes: **TokenEvent.java**, **TokenListener.java**, **VisionCapture.java** and **TokenManager.java**. The *VisionCapture.java* receives token actions from the server and generates *tokenEvent*. Instead of sending *tokenEvent* directly to the all *TokenListener* objects which are registered to receive the token event, *VisionCapture.java* writes the *tokenEvent* in to a map. Whenever system has time *TokenManager.java* reads the map for new *tokenEvent* and sends them to all *TokenListener* objects. To run *VisionCapture.java* the following method should be called in the default constructer of the application: ***TokenManager***.`startVisionCapture`() and it will initiate the a process as it is shown in Figure 6.

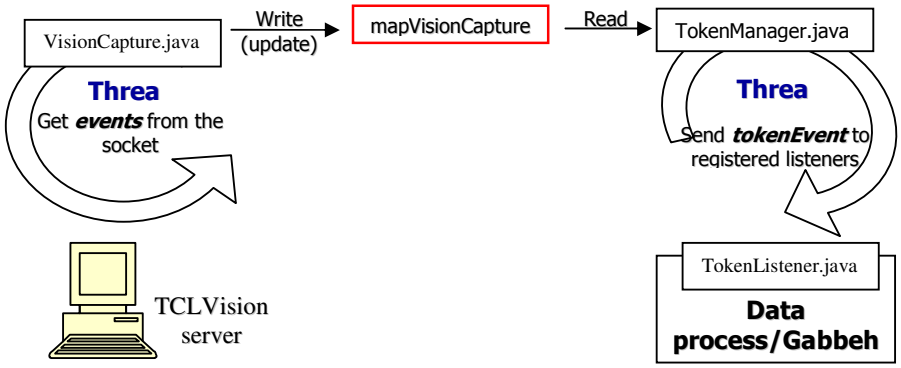


Figure 6: Generateing *tokenEvent* from token action

TokenListener.java is the listener interface for receiving token events. The methods in this class are empty. The classes which are interested in processing token events should implements this interface. The class that implements TokenListener interface should define all of methods of this interface. For example to create a listener object in *sheet.java* to receive token events and process them, the following code should be added:

```
class InternalTokenListener implements TokenListener{
      public void tokenAppear(TokenEvent e){}
      public void tokenMotion(TokenEvent e){}
      public void tokenDisappear(TokenEvent e){}   }
```

After creating the listener object, it has to be registered so it would receive generated *tokenEvent*. To register the listener object, it has to be added to list of objects which are interested to receive token events by calling *TokenManager* add method. Therefore to register the InternalTokenListener example, following code should be added to the default constructor *sheet.java*:

```
InternalTokenListener tokenListener = new InternalTokenListener();
TokenManager.add(tokenListener);
```

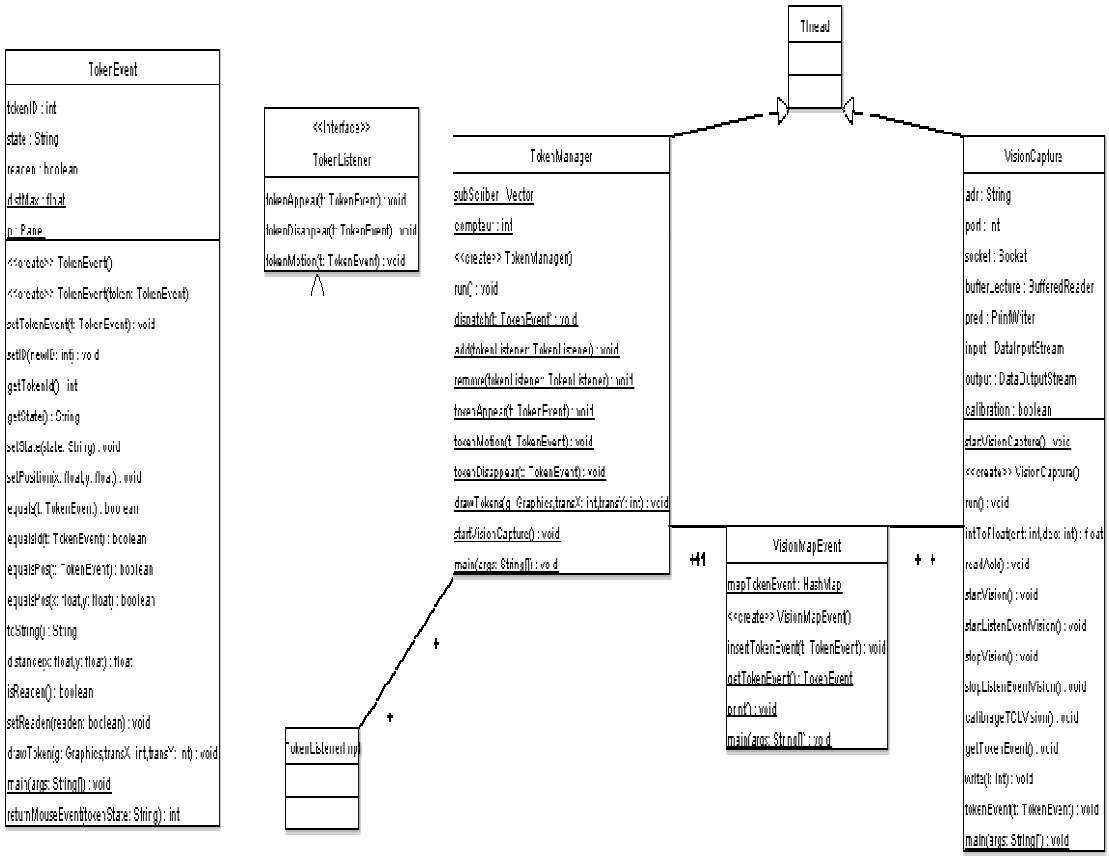


Figure 7 : Multi-Pointer class diagram

3 Simulator

The TCLVision and Augmented table are essential means to achieve the main aim of this work which is enabling Gabbeh to receive multi entries. Since this project is the result of the cooperation between two institutes and researches involved are spread across three locations and augmented table is only provided at one of these locations. Therefore it came to attention that developing a simulator for augmented table is necessarily since not all of the team members have direct access to the augmented table and it is important to be able to evaluate the design repetitively at early stages of development process. Also it would encourage a more iterative design process by making it easier to test every step of the design and development, since the simulator could be run on the same machine as the Gabbeh is running and it would save time and money which is needed for setting up and lunching TCLVision and augmented table for each single test.

The simulator is developed in way to be easy to use. It has a simple sheet which simulates the surface of the augmented table. The mouse is used to perform token actions on the sheet. Following mouse actions are used to simulate token actions on the augmented table: mouse left button click as token *appear*, mouse drag as token *motion* and mouse right button as token *disappear*. Developing the simulator offers two main advantages; first is to generate token actions which make it possible to validate Multi-Pointer-Vision package for generating *tokenEvent* and sending them to registered listener objects, and second is to make it possible to evaluate the use of Gabbeh tools (e.g. pen, comment tool, zoom slider) when a simulated token *appears* or it's location changes (token *motion*) on the simulator sheet.

Figure 8 demonstrate how a user is managing five tokens on the simulator's sheet. Token A is been used to adjust the zoom level. Token B is used to select the pen. And token C, D and E have been as three distinguished inputs for one selected tool.

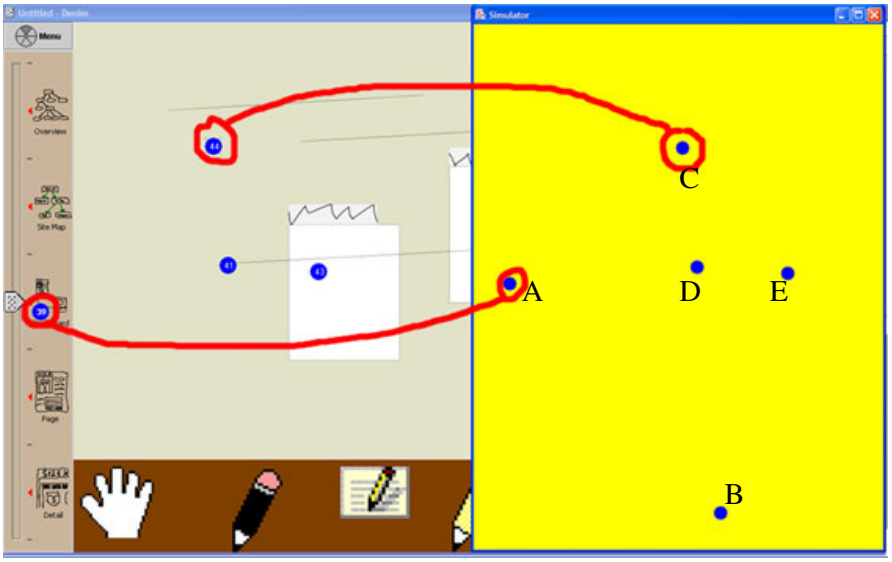


Figure 8 : Simulator of TCLVision.

Figure 9 shows a closer view of how the feedback of token is look like on *DenimSheet.* The number displayed on the feedback is same as the *tokenID* which helps in recognizing tokens when there is more than one user working on the table or the simulator.

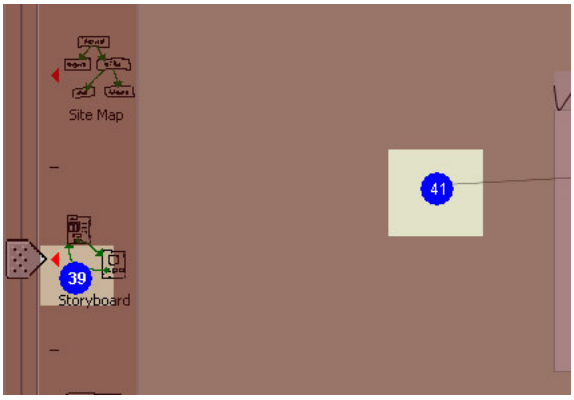


Figure 9: Token feedback on DenimSheet

4 Modifications

As it was explained in the introduction, Gabbeh is implemented by extending Denim and Satin projects. To enable Gabbeh to support multi entries some of the files of each underlying project have been modified which are explained in this section. This section is consisted of two subsections which explain the files which have been modified in Denim and Satin.

4.1 Modified Denim files

4.1.1 DenimConstants.java

When the *MouseListener* is used, it is possible to identify the relative position of the curser on the sheet by accessing *MouseEvent*. But when *TokenListener* is used, *TokenEvent* only store the absolute position of the token on the screen which is not the same as the relative position of token on the *DenimSheet*. Therefore to calculate the relative position of the token, it is important to find the absolute position of the *DenimSheet* on the screen. Then it is possible to check if the token is within the sheet and calculate the relative position of the token.
Since *DenimSheet* is not visible on the simulator window it is not possible to use `getPositionOnTheScreen` method as we use on the table. This would be same when token is placed on *ZoomSlider* or *ToolsArea*. Therefore the absolute position of each area is stored in *DenimConstants*:

```
public static final int DEFAULT_ZOOMBOX_WIDTH  = 100;
public static final int DEFAULT_SHEET_HEIGHT = 900;
```

These values (height of the sheet and width of zoombox) are used to insert the absolute position of interesting components when it is needed by calling *DenimConstants*. Now it is possible to check if *tokenEvent* is inside a component or not. For example to check if *tokenEvent* was occurred in *DenimSheet*, first *SatinConstants.DEFAULT_ZOOMBOX_WIDTH* is used to find the relative position of the *tokenEvent*,

```
tokenEvent.translatePoint ( - SatinConstants.DEFAULT_ZOOMBOX_WIDTH, 0);
```

Then check if relative position of tokenEvent is within the *DenimSheet* (parent):

```
if (tokenEvent.getX() < 0 || tokenEvent.getX() > parent.getWidth()){
        return;
}
if (tokenEvent.getY() < 0 || tokenEvent.getY() > parent.getHeight()){
        return;
}
```

To find relative position of *tokenEvent* in *ToolsArea*, both *DEFAULT_ZOOMBOX_WIDTH* and *DEFAULT_SHEET_HEIGHT* are required. It is explained in ToolsArea.javaTool.java section.

4.1.2 ToolsArea.java

ToolsArea is a container (JPanel) which contains all the available tools in Gabbeh. Each tool is placed in *ToolsArea* as a JLabel and could be selected by using a mouse click. When a tool is selected the mouse curser changes to the selected tool, for example when pen is selected, mouse curser would change to pen. Also pen would be disappeared from the *ToolsArea*. To select another tool, for example commenting-pen, the user could click on commenting-pen and the pen is dropped in the *ToolsArea* and curser changes to commenting-pen. It is not possible to perform this procedure when a token is used, since the curser is disabled and not visible when the *TokenListener* is used. Therefore it is not possible to identify which tool has

been selected, also tools size in tools area are not appropriate to be used on a big display such as augmented table, and it would be difficult for user to select the tool by using a token. To over come this problem large buttons have been used instead of labels to display the tools (see Figure 8). It is important to note that the token which is used to select the tool is different from the tokens which are used by users for designing. In the current version, only one tool can be selected that will be shared by all users. It means that one token (e.g. token B in Figure 8) is used to choose the interested tool and it will remain there while users are using that tool to perform other actions using other tokens (e.g. token D, E and C in Figure 8). Therefore it is important to check if the relative position of the *tokenEvent* is inside the tools area before processing it.

```
TokenEvent.translatePoint( -DenimConstants.DEFAULT_ZOOMBOX_WIDTH,
                           -DenimConstants.DEFAULT_SHEET_HEIGHT);
if (TokenEvent.getX() < 0 || TokenEvent.getX() > ToolsArea.getWidth()){
      return;
}
if (TokenEvent.getY() < 0 || TokenEvent.getY() > ToolsArea.getHeight()){
      return;
}
```

To enable the *ToolsArea.java* to receive and process *tokenEvent* the code has been modified and an *InternalTokenListener* was added. *InternalTokenListener* implements the *TokenListener* methods to enable the user choose the interested tool by using a token.

```
class InternalTokenListener implements TokenListener{
      public void tokenAppear(TokenEvent tmpEvt){}
      public void tokenMotion(TokenEvent tmpEvt){}
      public void tokenDisappear(TokenEvent tmpEvt){}
}
```

The *InternalTokenListener* has to be added to the object listener list for receiving *tokenEvent*, therefore the following line is added in the *ToolsArea* constructor:

```
TokenManager.add(new InternalTokenListener());
```

InternalTokenListener implements *tokenAppear* to enable the user to select an interested tool button by using a token *appear* action over that button. The *tokenAppear* method receives *tokenEvent* and translates the position of the *tokenEvent* to check if the token *appear* action was occurred within the tools area. If the position is valid, then it checks if any tool was previously in use. If it was then drop the old tool. The *getTool* method is called afterwards to check if the token is placed over the bounds of one of the tool buttons. The *getTool* method uses the *tokenEvent* position and returns the interested tool if the token is placed over any of the tool buttons and let the tool be grabbed by calling the *tool.grab* method.

```
TokenEvent evt = new TokenEvent(tmpEvt);
.. ..
.. .. check if the tokenEvent position is valid
.. ..
Tool currentTool = ui.getCurrentTool();
if (currentTool != null) {
      currentTool.drop(..,..);
}
Tool tmpTool = getTool(evt.getX(),evt.getY());
      if (tmpTool != null){
            tmpTool.grab();
}
.. ..
```

InternalTokenListener implements *tokenMotion* to enable users swap tools when a token is moved within *ToolsArea* from one tool position to another. The *tokenMotion* works in the same way as the *tokenAppear* method works. It checks the position of the *tokenEvent* for each *motion* action, and then if it is valid, it drops the current tool that is in use and grabs the tool which is returned by *getTool* method for that position.

To drop the tool user can perform token *disappear* action. *InternalTokenListener* implements *tokenDisappear* method to receive the *tokenEvent*, check if it has a valid position and *drop* the current tool that is in use.

When the simulator is in use, to enable the user identify which tool is selected *ToolsArea* has to paint a feedback on the selected tool. The current version of the code does not provide such feedback, as it can be seen in Figure 8 that only the token B that is placed within the tools area has no feedback.

4.1.3Tool.java

This class only encapsulates the generic behaviour of a tool and specific tools extend this abstract class. Two of the main methods in this class are *grab* and *drop* which control the selection of the tools as it was explained in the previous section. The code is modified to prevent the tool from being disappeared from the tools area and cursor changes when a tool is selected.

4.1.4 ZoomSlider.java

ZoomSlider is placed on the left hand side of *DenimSheet* and controls zoom level of the design model. To enable user to change the zoom level by using a token the *ZoomSlider* is modified and implements *TokenListener*. This enables *ZoomSlider* to receive *tokenEvent* and process it. It allows the user to perform token *appear* action to change the zoom level.
In addition *ZoomSlider* was modified to provide feedback on the zoom bar when a token is placed in the simulator, the *PaintComponent* method is modified to call *TokenManager* and draw the feedback on the slider.

```
TokenManager.drawTokens(g[3], TranslateX, TranslateY);
```

4.2 Modified Satin files

4.2.1 Sheet.java

Sheet is the main panel which contains all the screens and patches such as *DenimLabel*, *DenimPanel*, Comments and Strokes. *Sheet* is modified to implement *TokenListener* by adding an *InternalTokenListener*. This would set the focus to the *sheet* when user is drawing by a token on the *sheet*. The *MouseListener* is also been disabled since there is no need to have mouse input when Gabbeh is running on the augmented table or the simulator.
Also the paint method in *sheet.java* is modified to enable token feedback on the sheet. The paint method calls *TokenManager* to paint the tokens that have *appear* or *motion* status.

4.2.2 StrokeAssembler.java

StrokeAssembler gathers mouse events and creates a *stroke*. When it gets a mouse event, it can consume it (`AWTEvent.consume()`) and not let other classes use that event. If a mouse event is already consumed the StokeAssembler ignores it. *Stroke* is the main input action in the Satin application and there are three action events which are supported in Satin

[3] `g` - is the Graphics context to draw in

(*NewStrokeEvent*, *UpdateStrokeEvent* and *SingleStrokeEvent*). Following methods define the behaviour of the *GraphicalObject* that the *stroke* was drawn on it and one of the above stoke events was occurred in it.

```
handleNewStroke(NewStrokeEvent)
handleUpdateStroke(UpdateStrokeEvent)
handleSingleStroke(SingleStrokeEvent)
```

Before that the *strokeEvent* is dispatched to the interested *GraphicalObject* that was accrued on it, the *strokeEvent* is dispatched to the registered *Gesture Interpreter* and *Ink Interpreter* to get process as it is explained in Satin section.

To enable users generate *strokes* by using tokens, *StrokeAssembler* should accept *tokenEvent* and accumulates the *tokenEvents* to generate *stokes*. *StrokeAssembler* is modified to implements *TokenListener* in the same way that was explained in Multi-Pointer-Vision section. The position of each *tokenEvent* is checked to be within the *sheet* bounds and then the relevant method would process the event.

Since there are more than one token used on the table at the same time, it is important to be able to recognize the correct stroke to add the tokenEvent position. Therefore two hash maps are introduced to keep track of the relation between a token and a stroke.

```
public HashMap mapTimedStroke;
```

Since the token appears only once in the process of creating a stroke, in *tokenAppear* method the *tokenID* is used to start tracking of the token and the stroke:

```
String key = String.valueOf(tokenEvent.getTokenId());

mapTimedStroke.put(key, currentStroke);
```

The TokenMotion method is called in the process of creating a stroke as long the token is moving. For each token move action, tokenMotion method is implemented in following ways to find the relevant stroke to continue the process:

```
currentStroke = (TimedStroke)mapTimedStroke.get(key);
if (currentStroke == null) {
      return;
}
```

Then it continues the process and adds the updated stroke to the map at end of the method with using same key. It is important to note that the *tokenID* is the ID for each individual token and it does not change when different actions are performed with the same token.

When the token is disappeared, the key and timed stroke gets removed from the map and Timed Stroke is generated.

```
mapTimedCurvyStroke.remove(key);
```

4.2.3SatinConstants.java

SatinConstants is modified in the same way that *DenimConstants* is modified and is explained in DenimConstants.java section.

4.2.4 StrokeEvent.java

StrokeEvent is the interface class for all the events generated for a stroke. The strokeEvent is modified to include two methods to track which token has generated the action for an interested strokeEvent.

```
public void setTokenEvent(TokenEvent evt)
public TokenEvent getTokenEvent()
```

4.2.5 ImmediateInkFeedBackInterpreter.java

Since the StrokeAssembler is modified to handle more than one input at the same time, it is important to modify all the classes that *strokeEvent* is dispatched to them, (such as *Gesture Interpreters* and *Ink Interpreters*) to enable them to track *tokenEvents* for different token at the same time. Therefore *ImmidiateInkFeedBackInterpreter* has been modified include a map containing *tokenID* and *strokeEvent* in a similar way that it was modified in *StrokeAssembler* class.

In the current version, we only have modified *ImmidiateInkFeedBackInterpreter* from all the available interpreters. This is because *ImmidiateInkFeedBackInterpreter* is being used in most of the tools and it has a direct effect on the way that Gabbeh is used. Modifying this interpreter prevents confusions between different tokens and not paining unwanted stroke lines between different tokens.

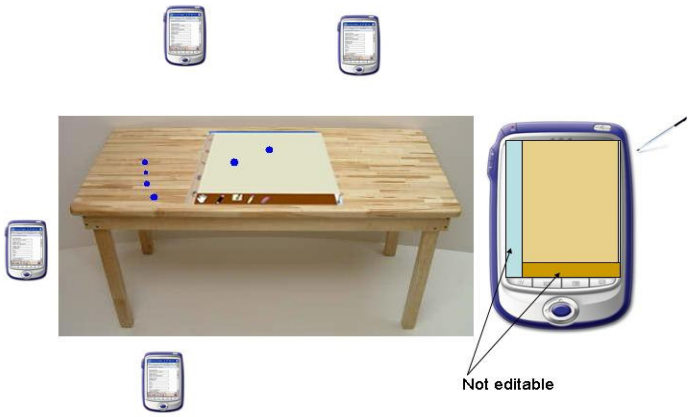
5 Further Design development

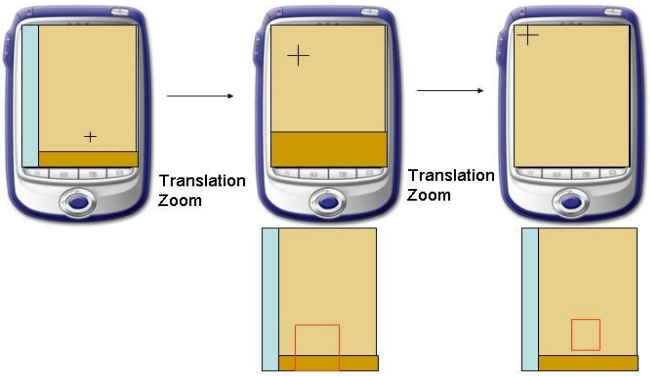
5.1 Simulator

To improve the testing it is recommended to implement the simulator as part of Gabbeh by modifying the MouseListener. This could simulate a closer feel and look to what happens on the table by having the simulator running on top of the Gabbeh (the design has not discussed yet).

5.2 PDA – Tablet PC

Token size makes it inappropriate to be used for designing details such as handwritings and low level sketches. A possible way to encourage more user involvement and enabling designers to sketch low level details is to use portable devices which support tablet technology and can be used around the augmented table. Since devices like PDA does not have large enough displays, it is not possible to display complete design sheet in their screens. Therefore one possible solution is to let user to have a abstract view of the design sheet and select a point on his/her PDA and then the client version of Gabbeh on the PDA let user to zoom in to that point and make changes. It also would let users to use the attached keyboard to PDA or the Tablet PC to use for entering text since it is not possible to enter text by using tokens on the table!





5.3 Voice comments

Recording speech, voice comments.

5.4 Finger tracking

Update the current version of Multi-Pointer-Vision with the work of Julien Letessier in order to use directly fingers.

6 Further Code Developments

6.1 Fix Current Bugs

6.1.1 Move a page

There is bug which cause the page gets deleted when one user is moving it and another user appears a token on the table. This is important to fix this problem since interrupt the collaboration when two users want to move two pages. It is expected that the source of the problem is in the SelectAndMoveInterpreter.java. It is required to modify this class to map tokenEvent to tokenID.

6.1.2 DenimPanel and Interpreters

Modify DenimPanel to implement *TokenListener*. This would enable a panel receive tokenEvent and also to associate a panel to a token. (it is important for future work .. more thinking)
Also it is important to modify all existing interpreters to implement TokenListener and map each tokenEvent to its tokenID. This is to enable more than one token be used on the table without any conflict between them. It has not been done because of the time limitation and should be a straight forward modification to do.

6.2 Comment panel

To enter a text comment or to manage the colour and visibility of comments user needs to on the CommentDialogBox. The CommentDialogBox interface is not suitable to be used with token as input. The Interface needs to change in away which would enable the user change the colour of comment by using a token.

6.3 MouseListener/TokenListener

It is important to be able to switch between MouseListener and TokenListener. Since MouseListener conflict with TokenListener when Gabbeh is running, MouseListener is disabled wherever that TokenListener is enabled. It is required to have an option to make it possible to choose which listener is required to work with. Since Gabbeh is not always going to be run on a table or just on a pc.

6.4 Many tools

In the current version of this work, all the users can share only one tool at time. It means when pen is selected, all the users at the table can only work with pen. And every token which appears on the table functions as the pen. It is important to enable different users to use different tools at the same time in such collaboration. One user might want to move a page while the other one wants to create a new page. These different actions require different tool

to be associated to different tokens. A possible way of overcoming this limitation is to introduce a toolbox for every token that appears on the sheet. Each token would have its own toolbox and can select the tool it wants to use. Or all tokens can share same toolbox but each token would have an attribute that stores the tool that token represents. However it is important to identify which tools can not be used at same time before hand. It is mainly because each tool is associated with a interpreter and it is important to know if two interpreter can be used at the same time or not.

.