Rapidly Prototyping Single Display Groupware through the SDGToolkit

Edward Tse and Saul Greenberg
Department of Computing Science,
University of Calgary, Alberta, Canada T2N 1N4
Email: [tsee, saul]@cpsc.ucalgary.ca

ABSTRACT
Researchers in Single Display Groupware (SDG) explore how multiple users share a single display such as a computer monitor, a large wall display, or an electronic tabletop display. Yet today's personal computers are designed with the assumption that one person interacts with the display at a time. Thus researchers and programmers face considerable hurdles if they wish to develop SDG. Our solution is the SDGToolkit, a toolkit for rapidly prototyping SDG. SDGToolkit automatically captures and manages multiple mice and keyboards, and presents them to the programmer as uniquely identified input events relative to either the whole screen or a particular window. It transparently provides multiple cursors, one for each mouse. To handle orientation issues for tabletop displays (i.e., people seated across from one another), programmers can specify a participant's seating angle, which automatically rotates the cursor and translates input coordinates so the mouse behaves correctly. Finally, SDGToolkit provides an SDG-aware widget class layer that significantly eases how programmers create novel graphical components that recognize and respond to multiple inputs.

KEYWORDS: Single display groupware, interface toolkits, co-located collaboration, groupware architectures, CSCW.

INTRODUCTION

Researchers in Computer Supported Cooperative Work (CSCW) are now paying considerable attention to the design of *single display groupware* (SDG) i.e., applications that support the work of co-located groups over a physically shared display [12]. What distinguishes full SDG from conventional windowing systems is that each participant has his or her own input device, allowing all to interact simultaneously with the common display.

Sporadic research in SDG began over a decade ago with the demonstration of the MMM system [2], followed by technical explorations of SDG architectures [e.g., 11, 1], SDG interaction methods [e.g., 12, 13] and many studies of how children share a display in educational settings [e.g., 5, 10]. Recently, SDG has surged in importance due to the opportunities presented by projectors and other large displays that can be attached to walls and/or used horizontally as electronic tabletops.

The problem is that SDG is still notoriously hard to build. Typically, researchers develop their own specialized applications from the ground up, resulting in SDG that is tedious to implement, difficult to maintain and modify, and tough for other researchers to replicate. While most researchers are interested in interface design issues and SDG use, excessive effort is spent developing the underlying plumbing. This problem is exacerbated by our current generation of windowing systems that make it difficult to do even the most basic SDG activities:

- ***Multiple input and identification:*** There is no convenient way to gain and uniquely identify the multiple input streams from mice and keyboards.
- ***Multiple cursors:*** Systems supply only a single cursor. Yet almost all SDG applications require multiple cursors, one for each attached mouse.
- ***Table orientation:*** Tabletop developers face considerable hurdles circumventing orientation issues that occur when end users are seated at different sides of the table, e.g., how the cursor appears, how the mouse behaves, how coordinates are handled.
- ***SDG user controls***: Conventional controls (*aka* widgets) such as buttons, menus and even windows cannot distinguish which SDG user interacted with it, only store a single input focus between them, and are not designed to handle concurrent use.

Our own frustrating experiences in SDG echoed these problems. We began developing SDG interface widgets [13] with the MID (*multiple input devices)* toolkit [1], but had to abandon it as it worked only with Windows 98. It then proved impossible to get individual mice and keyboard streams from the later Windows 2000 and NT systems. Seeking alternatives to the mouse, we developed PDA-based input devices [7; *see also* 11], and even rewrote the firmware of a USB mouse so that the window system saw it as a Phidget (a physical widget) instead of a mouse [8]. Even then, coordinate tracking and cursor drawing was painful and inefficient. Especially disconcerting was that our time and effort went into infrastructure development *vs.* our main focus: the design and evaluation of SDG interaction techniques over upright displays and tabletops.

Cite as:
Tse, E. and Greenberg, S. (2003) **Rapidly Prototyping Single Display Groupware through the SDGToolkit.** Report 2003-721-24, Department of Computer Science, University of Calgary, Calgary, Alberta Canada. March.

Consequently, we decided to design and build a toolkit that would help us and others rapidly develop SDG applications and interface components suitable for upright displays and tabletops. Our driving goal was that the toolkit would be:

- simple enough for average programmers to quickly learn and use, where they can concentrate on SDG application design rather than low level SDG plumbing.

The result is SDGToolkit, and this paper reports our experiences. We begin by presenting the fundamental problems in SDG development, how the SDGToolkit architecture solves them, and how the end-programmer sees these solutions. Next, we illustrate what the end-programmer would have to do to create a few simple SDG applications. The subsequent section is concerned with infrastructure for creating true SDG-aware user controls (widgets). This is followed by example applications and SDG widgets built with the toolkit. We conclude by relating our work to other SDG systems, especially the MID multiple input devices toolkit [1].

While this paper describes what some may consider 'routine' software development, we stress our contributions have a much broader impact to SDG research. Specifically:

1. We articulate the basic requirements and technical challenges that face all designers of single display groupware toolkits. This is important as it helps others understand the needs and pitfalls in SDG development *a priori* rather than by after-the-fact discoveries through trial and error.
2. We detail solutions to these problems as implemented in SDGToolkit. While our descriptions are within the context of the Microsoft Windows platform and .NET, our strategies would generalize to other platforms and thus help other developers of SDG toolkits.
3. We describe how end-programmers would process and use SDG input events, and how they would develop and/or use SDG widgets. This is important as it supplies a conceptual model to other toolkit builders about how a toolkit for SDG should present itself.
4. We provide SDGToolkit as a fully documented downloadable resource for others so they can immediately begin SDG research.

SDG TOOLKIT – FUNCTIONALITY & ARCHITECTURE

By definition, SDG allows the simultaneous use of multiple input devices. Consequently, a basic SDG toolkit must address requirements and technical challenges fundamental to managing multiple mice and keyboards. In this section, we describe various technical SDG challenges in turn, and explain how our SDGToolkit implements each solution. Figure 1 is our anchor: it shows the SDGToolkit class and event architecture, and we will use it to illustrate how the various pieces fit together. We again emphasize that while our toolkit is based upon Windows and .NET, our general approach to solving these SDG challenges are replicable in most windowing systems.

Note on terminology. Controls, user interface components, and widgets are used synonymously, as are windows and forms. We refer to mice as synonyms for pointing devices (pens, digitizing tablets…) and fully expect future versions of our toolkit to include novel pointing devices such multiple touch surfaces, e.g., MERL Diamond Touch [4], and Smart Technologies DViT [www.smarttech.com].

Gaining the Device Input Stream

For anything to work in an SDG setting, we have to discover what pointing devices and keyboards are attached to the computer and identify a separate input stream for each one. While this should be simple, in practice most windowing systems present significant hurdles because of the special way they deal with the system mouse and keyboard. The first problem is that all windowing systems combine the input from multiple mice and keyboards into a single system mouse and single keyboard input stream. For example, if two USB mice were attached to a computer, and if these mice were moved left and upwards respectively, the merged stream would move the cursor diagonally up and left. Only this combined stream is easily available to the programmer.[1] The second problem is that non-standard input devices (e.g., game controllers, joysticks, digitizing tablets) at their worst require that the programmer write very low level code such as device drivers, and at their best requires one to use APIs (such as Microsoft's DirectInput) that do not interoperate well with the windowing system.

Solution. Windows XP introduced *Raw Input*, a somewhat difficult-to-program utility for low-level management of input. Programmers can query Raw Input to gain a list of all attached input devices. On any keyboard or mouse input, Raw Input adds it to a generic input stream, which the programmer can parse to identify what device generated that input and its particular arguments. For example, Row 1 of Figure 1 illustrates a Raw Input event stream. Each event is tagged by a handle identifying the input port, the input device type (e.g., mouse, keyboard), and its parameters.

SDGToolkit uses Raw Input as the building block for handling input from multiple keyboards and mice. In particular, SDGToolkit supplies the *SDGManager* class (the box contained between Rows 2 - 6) that captures, transforms and wraps the Raw Input into a more convenient form (Rows 2 - 4). When the programmer creates the SDGManager instance, it queries Raw Input (Row 1) to discover the attached mice and keyboards. The SDGManager then automatically creates instances of the SDG Mouse and Keyboard classes (Row 4), each matched to a particular input device by storing its handle (Row 4). Finally, the SDGManager parses the incoming raw input stream (operation in Row 2), and stores the mice/keyboard data in the appropriate Mouse and Keyboard instances

[1] The MID toolkit [1] used Microsoft's DirectInput to gain individual mice inputs in Windows 98. Unfortunately, Windows 2000 turned off this mouse access, compromising MID's utility for SDG.

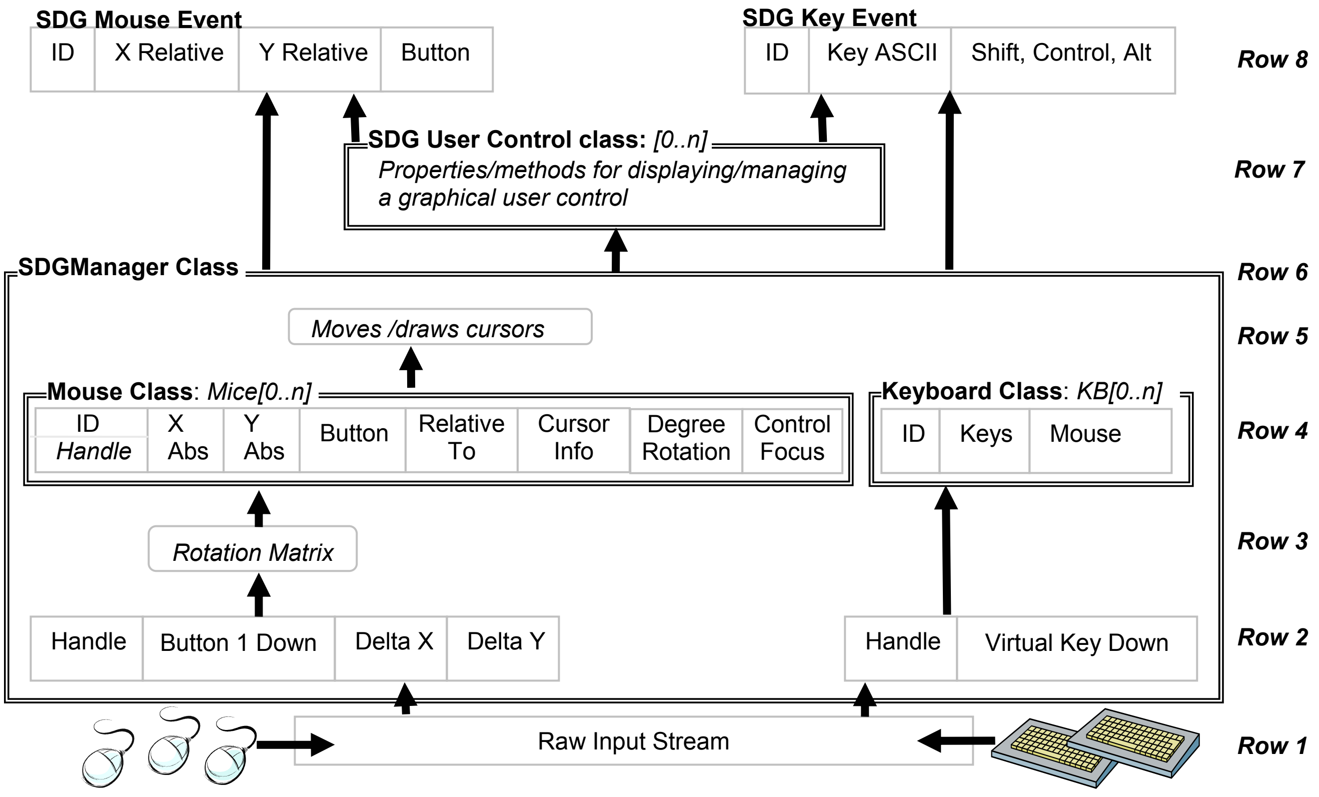


Figure 1. The SDG Class and event structure, showing how raw input turns into SDG events

(Row 4). We note that this is a general strategy: we can use the same approach to extend SDGToolkit to handle other types of input devices.

Furthermore, the SDGManager maintains this collection of all Mouse and Keyboard instances. Thus the programmer can easily find out how many devices of a particular type are attached and enumerate through them. For example:

```
// Initial mice positions: move all to 0,0
foreach (Mouse this_mouse in sdgMgr.Mice) {
   this_mouse.X = 0; this_mouse.Y = 0;
}
```

The SDGManager also generates ID's for each device instance as ordinal integers, starting at 0. This means that programmers can use this ID to index the SDGManager's Mouse and Keyboard collection, where they can easily query or set the properties of a particular instance. For example, we can display the coordinates of the 1st mouse by

```
Console.Writeln (sdgMgr.Mice[0].X + "," +
                 sdgMgr.Mice[0].Y;
```

Uniquely Identified Input Events

When a programmer receives an input event from an SDG toolkit, he or she needs to know which of the mice or keyboards generated that event. Traditional mouse and key event handlers do not provide this information.

Solution. As the SDGManager stores the data in a particular mouse/keyboard instance, it also raises an *SDG Mouse Event* or *SDG Key Event*, which is presented to programmers in a style that mimics standard mouse and keyboard events (Figure 1, Row 8). For example, SDG Mouse Events follow the standard `MouseDown`, `MouseUp`, `MouseMove` and `MouseClick` naming conventions, and it contains all the expected parameters, e.g., X and Y coordinates, button state, and so on. Similarly, the SDG Key Events include `KeyUp`, `KeyDown` and `KeyPress`. The major and very important difference from standard events is that we add the ID parameter into all events arguments class (Row 8). The result is that programmers can create event handlers that easily identify the mouse or keyboard that fired the event.

For example, Figure 2 compares how a C# programmer would register and write a standard non-SDG mouse event handler (Figure 2 top) *vs.* an SDG mouse event handler[2] (Figure 2 bottom). The important differences are the inclusion of a mouse ID, the different typing of the event argument (`SdgMouseEventArgs e`) and that the SDGManager generated the event (`sdgMgr.MouseDown`) instead of the window (`Form.MouseDown`).

Translating Pointer Data to Window Coordinates

In traditional graphical user interface programming, mouse pointer events are generated by the active window or control, and all coordinates are returned relative to it. This is very convenient because it is this active window/control that is the programmer's usual context for interpreting events and/or for drawing graphics. Within an SDG toolkit,

[2] While examples are in C#, SDGToolkit works with any .NET language e.g., Visual Basic, Managed C++ and so on.

```
// a traditional mouse event
Form.MouseDown += new MouseEventHandler(OnMouseDown);
…
OnMouseDown (object sender, MouseEventArgs e)  {
    Console.Writeln ("X,Y,button is: "
                     + e.X + e.Y + e.Button);  }

// an SDG mouse event – differences are bolded
sdgMgr.MouseDown += new MouseEventHandler(OnMouseDown);
…
OnMouseDown (object sender, SdgMouseEventArgs e)    {
    Console.Writeln ("ID, X, Y, button is:"
                     + e.ID + e.X + e.Y + e.Button);}
```

Figure 2. Comparing traditional and SDG mouse events

we would like to do the same thing. However, pointing devices usually deliver only delta values relative to their last movement to the low level input handler. For example, Raw Input's event stream reports mouse movements as +/- some increment e.g., (+2, -1). While converting this to window coordinates should be straightforward, traditional controls (such as a top level window or even a button) do not generate SDG mice events, and thus we do not know the context of where our SDG events occurred. This is why the SDGToolkit example in Figure 2 bottom has the SDGManager deliver SDG events instead of the `Form` window (as in the top example of Figure 2).

Solution. By default, we transform Raw Input delta values into absolute screen coordinates that are stored in the SDG Mouse Instance (Figure 1, Row 4). Unless otherwise instructed, the SDGManager includes these screen coordinates whenever it raises an SDG Mouse Event.

Because screen coordinates can be unwieldy, we let programmers explicitly associate mouse instances to both standard windows and controls. Specifically, they set the Mouse's `RelativeTo` property to the desired window/widget; the SDGManager will now translate and return the mouse coordinate relative to that window or user control (Figure 1, Row 4; see Mouse Class). For example,

```
  SDGMgr.Mice[0].RelativeTo = Form;
```

instructs the SDGManager to return coordinates for the first mouse relative to the `Form` top level window instead of as screen coordinates. Because the SDGManager does the coordinate transformation on the fly at run time, the RelativeTo property can be changed any time during program execution.

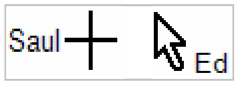
In a later section, we will describe how our SDG User Control class define controls that receive events from the SDGManager, and how these controls automatically translate the event screen coordinates to control-relative coordinates. The SDG control then re-raises these modified events (Figure 1, Rows 7+8). This is now identical to how windows and controls raise events in the traditional programming model shown in the top of Figure 2.

Displaying Multiple Cursors

In single user systems, programmers expect to get cursors for free, where the cursor moves fluidly as it responds to pointer movements. The problem for SDG developers is that our standard operating systems provide only one cursor, and we need multiple cursors representing each pointing device. In addition, we need the ability to visually distinguish between these cursors. While implementing multiple cursors is a straight-forward graphics problem, it can be very tedious for the SDG programmer to implement them at the application level if he or she wanted to avoid drawing artifacts while still maintaining performance.

Solution. By default, every pointing device seen by SDGToolkit displays an associated cursor. No extra end-programming is needed to get these basic multiple cursors. The SDGManager is responsible for this (Figure 1, Row 5). It implements it by leveraging the capabilities of top-level transparent windows, where one is created for each Mouse instance. SDGManager draws the cursor within this window, and repositions the window after each mouse move to the correct position. As long as cursors are of modest size, they perform well, especially if the computer uses video cards that process transparent windows in hardware.

SDGToolkit cursors are also highly customizable. The programmer can set the various cursor properties contained in each mouse instance (Figure 1, Row 4) to redefine the cursor shape, its hot spot, whether it is visible, and even its transparency. The programmer can also add a text label to the cursor, and can adjust the text font, size, color and location relative to the cursor graphic. For example, the following code snippet creates these two visually distinctive cursors identified by their owner's name.



```
SDGMgr.Mice[0].Cursor = Cursors.Cross;
SDGMgr.Mice[0].Text   = "Saul";
SDGMgr.Mice[0].TextCardinalPosition = West;
SDGMgr.Mice[1].Cursor = Cursors.Arrow;
SDGMgr.Mice[1].Text   = "Ed";
```

Supporting both Tabletop and Vertical Displays

While almost all early work on SDG was on traditional monitors and electronic whiteboards, recent work has focused on horizontal displays such as electronic tables. Unlike upright displays, users are often seated in many different orientations around a table, e.g., 'kitty-corner', facing one another, side by side, etc. The problem is that mouse movements and cursor appearance always assume a single orientation; thus from any but the 'South' person's perspective the cursor and text labels will be oriented incorrectly, and the mouse is unusable as it seems to move in the wrong direction.

Solution. The programmer can set an orientation for any mouse through the SDGManager using the Mouse instance's `DegreeRotation` property (Figure 1, Row 4). The mouse cursor and mouse movements are adjusted accordingly to give the cursor the correct look and the mouse the correct feel. For example, if one person is sitting across from the other, we would set `DegreeRotation` to 180; the cursor and text caption would be flipped 180 degrees, and cursor movements would inverted. For simplicity, the SDGManager does this coordinate transformation directly on the deltas produced by Raw Input through a rotation matrix (Figure 1, Row 3). Finally, the SDGToolkit also adjusts the rotated cursor so that it will always appear on-screen. All this dramatically simplifies tabletop programming, as the SDG Toolkit takes care of all translation, rotation and cursor resizing issues.

Dealing with the System Mouse

The next technical challenge is an artifact caused by the way current windowing systems interpret the system mouse. The problem is that there is only one true system mouse. Recall that a standard window system merges multiple pointer inputs to move this single system cursor. Consequently, this system mouse is still moving around the screen as it responds to all mouse movements, even if we turn off the display of its cursor. This leads to quandaries for SDG developers in terms of how they manage this system mouse. We present these problems, but forewarn that there are no elegant solutions. Instead, we list various approaches we could take and show how each mitigates problems caused by the system mouse.

First, if all SDG mice move the system mouse, it will not track correctly (as it reacts to the combined forces on it): it will appear as an extra cursor moving around the screen in strange ways. While we could make it invisible, it is still active i.e., a click with any SDG mouse also generates a click on the system mouse: this could mysteriously activate the window or widget under the system mouse.

One possible solution is to continuously move the system mouse to the location of the most recently used SDG mouse i.e., to give the momentary illusion that any SDG mouse could control a non-SDG window or control. Unfortunately, this does not work well in practice. Time and location dependencies in how a system mouse interpreted concurrent click/move/release actions generated by multiple mice meant that one user's mouse action could easily interfere with another user's mouse action.

A much better solution is to bind the system mouse to directly follow a single SDG mouse and its cursor. This 'super mouse' will have both SDG and standard capabilities. While not a democratic solution, it is pragmatic. This solution is implemented by SDGToolkit, where the programmer can ask the SDGManager to bind the system mouse to a single SDG mouse, for example:

```
SDGMgr.MouseToFollow = 1;  //follow mouse #1
```

However, a serious side effect of having an enabled system mouse results from windowing systems maintaining only a single active window as the input focus. A super mouse click outside the SDG window causes the system mouse to raise a non-SDG window and the SDG application will lose the input focus. Other SDG mice will no longer respond.

To remove this side effect, we can 'turn off' the system mouse. We can do this by ensuring that it never moves from some unused corner of a window and by making it invisible. We also include this approach in SDGToolkit, where programmers set the `ParkSystemMouseLocation` property of the SDGManager. While excellent for managing pure SDG applications, it means that the end user cannot use standard window controls (close, resize), any standard widgets (buttons, scrollbars), or switch to other non-SDG windows. This can be confusing because people's naïve conceptual model is that their cursor represents both an SDG mouse and a system mouse.

Still, it would still be convenient if we could exploit non-SDG widget capabilities with a parked mouse. To do this, we can tell the program what widget appears under an SDG mouse event. In particular, whenever the SDGManager sees a mouse event, it examines what user control (if any) is immediately under that coordinate position. It then returns it as the `sender` argument e.g., as shown in Figure 2:

```
OnMouseDown(object sender, SdgMouseEventArgs e);
```

Of course, this does not completely solve the problem as it remains the programmer's responsibility to activate any of that widget functions. For example, if a user clicked over a non-SDG button then the programmer could identify this button in the `sender` argument and use it to interpret the event within the context of the button. However, the programmer would have to somehow activate the button – its graphical behavior and its callback - as the button has never received this event.

The choice between the solutions implemented by SDGManager – the single 'super mouse', mouse parking, using the sender argument – is a tradeoff between the desired nuances of the SDG application and its effect on the end user audience.

Managing Multiple Keyboard Focus

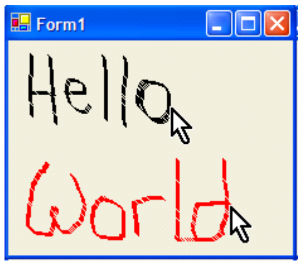
In a standard application, pressing a key on a keyboard usually associates that key event with a single input focus, i.e., the window or control where the character should be written and the event reported. Users change this focus by tabbing or by clicking into a text control with the mouse. The problem in SDG is that there can easily be multiple input foci, where each user of an SDG application may want text to appear in (say) a different text control.

Solution. We track multiple text foci for all keyboards and mice as follows. First, we associate each keyboard with a mouse. Second, when a user mouse-clicks over a control to indicate their text input focus, the Mouse instance automatically stores a pointer to that control in its `ControlFocus` property (Figure 1, Row 4). Third, the programmer writes a keyboard key event handler that just looks up the `ControlFocus` of its corresponding mouse and directs the text towards that control.

By default, Keyboard 0 is automatically mapped to Mouse 0, Keyboard 1 to Mouse 1 and so forth. Programmers can customize this mapping by changing the 'mouse' property of the keyboard instance e.g.,

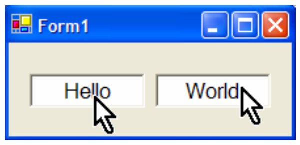
```
SDGMgr.Keyboard(5).Mouse = 0
```

causes the 6th keyboard to track the first mouse.



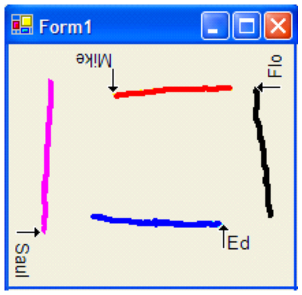
```
1  private void InitializeComponent () {
2   Form1.sdgMgr.RelativeTo = this;  //'this' refers to the top level window
form
3   this.sdgMgr.MouseMove +=new SdgMouseEventHandler(this.sdgMgr_MouseMove);
4    …
5  }
6  private void sdgMgr_MouseMove(object sender, SdgMouseEventArgs e) {
7    Graphics g = this.CreateGraphics();
8    Pen penColour = Pens.Black;
9    if (e.ID > 0) penColour = Pens.Red;
10   if ((e.Button & MouseButtons.Left) > 0)
11     g.DrawLine(penColour, new Point(e.X-1, e.Y-1),new Point(e.X+1, e.Y+1));
12 }
```

Figure 3a. SDG Hello World Drawing – 'Hello' is in black, 'World' is in red.



```
1  private void sdgMgr_KeyPress(object sender, SdgKeyPressEventArgs e){
2    TextBox this_textbox;
3    if (sdgMgr.Mice[e.ID].ControlFocus is TextBox) {
4      this_textbox = (TextBox) sdgMgr.Mice[e.ID].ControlFocus;
5      this_textbox.Text = this_textbox.Text + e.KeyChar.ToString();
6  }}
```

Figure 3b. SDG Hello World Keyboard



```
1 Public Form1 (){  // The Form constructor
2   String[] sdgText = {"Ed", "Saul", "Mike", "Flo"};
3   int[]    sdgDegreeRotations = {0, 90, 180, 270};
4   for (int i=0; i < sdgMgr.Mice.Count && i < 4; ++i) {
5     sdgMgr.Mice[i].Cursor = Cursors.UpArrow;
6     sdgMgr.Mice[i].Text = sdgText[i];
7     sdgMgr.Mice[i].DegreeRotation = sdgDegreeRotations[i];
8   }}
9 private void sdgMgr_MouseMove(object sender, SdgMouseEventArgs e) {
10  Graphics g = this.CreateGraphics();
11  Color[] colors = {Color.Blue, Color.Magenta, Color.Red, Color.Black};
12  if((int)(e.Button & MouseButtons.Left) > 0) {
13     Graphics g = this.CreateGraphics();
14     g.DrawLine(new Pen(colors[e.ID]),
15                new Point(e.X-1, e.Y-1), new Point (e.X+1, e.Y+1));
16  }}
```

Figure 3c. SDG Tabletop drawing. All user marks are in different colors.

WHAT THE PROGRAMMER SEES

This section illustrates how a programmer would actually create SDG applications with the SDGToolkit. For clarity, our examples are deliberately simple to minimize non-SDG complexity. Code excludes setup and housekeeping code standard to all SDG and non-SDG Windows programs.

Hello world – mouse drawing. Our first 'hello world' example is a very simple concurrent drawing application involving two users and two mice, illustrated in Figure 3a. It illustrates how SDG mouse events are handled. To build this, the programmer takes the following steps.

1. Using the Visual Studio interface builder, drag an SDGManager component from the Visual Studio toolbox onto the application.[3]
2. In the standard InitializeComponent routine (Figure 3a, line 1) that initializes the top level window, add two lines of code that first sets the `relativeTo` property of the SDGManager to the form (line 2), and then register an event handler to the SDG MouseMove event (line 3). Alternatively, one can set the event handler without coding by using the SDGManager's property window.
3. Write the callback for the `sdgMgr_MouseMove` event (lines 6-12). Create a black drawing pen (line 8), but change its color to red if the Mouse ID is greater than 0 i.e., if its not the first mouse (line 9). We then check to see if the left button is depressed for that mouse (line 10), and if so draw a 2x2 pixel around the current X and Y coordinates of the mouse (line 11).

These few lines of code illustrate the simplicity of the SDGToolkit. In contrast, building the same program without the SDGToolkit (atop of Raw Input) is an order of magnitude larger and certainly more complex!

Hello world – keyboard text. Our second 'hello world' example has two textboxes and also works with two people (Figure 3b). When a user clicks on a textbox, that user's keyboard `KeyPress` event will go to it. If two people click different textboxes as in Figure 3b, their typing will be

[3] The SDGManager is implemented as a non-visible control used by the programmer in exactly the same way as other standard controls. One adds it to a window by drag 'n drop, sets its many properties and event handlers through form-filling, and handles events in the normal way.

directed appropriately (even if they type simultaneously). If both click the same text box, their input is merged.

The code in Figure 3b shows only the `KeyPress` event handler, which illustrates how one associates `KeyPress` events from multiple keyboards to the different text widget foci. The logic is simple. Recall from the previous section that each mouse remembers what control it last clicked (the focus) in its `ControlFocus` property. When the `KeyPress` event is raised from either keyboard, the event handler (lines 1-6) finds the corresponding mouse (via the matching Id), checks to see if its `ControlFocus` property holds a Textbox control (line 3), and if so assigns it to a temporary variable (line 4). It then inserts the key character into this Textbox (line 5).

Tabletop drawing. Our third example illustrates a drawing application designed for a square tabletop with four seated people, one per side. As Figure 3c shows, cursors and text labels are oriented appropriately. What is not visible is that the person's mouse will also behave correctly given their orientation. The initialization code shows how the programmer deals with an unknown number of mice (up to 4 in this example – line 4), sets mouse properties such as cursors and their text labels (lines 5–6), and correctly orient the cursors and returned coordinates (line 7). The `MouseMove` event handler (lines 9-16) is very similar to Figure 3a, except that it shows a better way to assign different line colors to each user.

SDG USER CONTROLS

Programmers can now use SDGToolkit to easily create vertical or tabletop SDG canvases that respond to multiple input events. However, they still face considerable hurdles equipping these canvases with interaction controls, such as SDG-aware analogues to single-user buttons, menus, textboxes, palettes, and so on. Standard widgets are inadequate. They do not understand multiple input devices. They cannot deal with concurrent access correctly as they still maintain their single user semantics.

Consequently, we argue that any SDG toolkit must supply the following features to ease the end programmers' task of equipping SDG applications with appropriate controls.

1. Provide building blocks that let programmers create novel SDG controls exhibiting SDG semantics.
2. Controls include an event mechanism so that they can pass through SDG events for direct use by the end-programmer.
3. Include a stock set of useful SDG controls that a programmer can use immediately within an application.

This section describes how SDGToolkit includes these capabilities.

The SDG Control Interface

We began by creating two class interfaces that defined the minimum set of capabilities that any SDG control should understand. The `ISdgMouseWidget` interface defines the mouse capabilities, where we insist that any SDG control object must implement methods (with arguments) corresponding to the four normal SDG mouse events described in the previous section e.g., `OnSdgMouseDown`, `OnSdgMouseMove`, `OnSdgMouseUp`, `OnSdgMouseClick`. For example:

```
  void OnSdgMouseMove(SdgMouseEventArgs e);
```

The second `ISdgMouseAndKeyWidget` interface extends this interface to include the key events `OnSdgKeyDown`, `OnSdgKeyPress`, and `OnSdgKeyUp`. For example:

```
  void OnSdgKeyDown(SdgKeyEventArgs e)
```

If graphical controls on the screen contain these methods, then the SDGManager can exploit them to make them SDG-aware. In particular, whenever the SDGManager gets an SDG Mouse event, it looks for a control immediately under the mouse coordinate to see if it has these methods. If it does, then the SDGManager invokes those methods, passing through the appropriate arguments. Row 7 of Figure 1 illustrates this with a generic graphical control called 'SDG User Control class', discussed next.

The SDG User Control

While the above interfaces help provide the mechanism underlying SDG widgets, they are still too low level to be convenient building blocks for an SDG widget developer. Consequently, we give the SDG widget developer an inheritable object that has all the expected behaviors of a widget, and that implements the basic SDG interface.

Microsoft .NET supplies special objects called `Controls` and `UserControls` that are the building blocks for all conventional widgets. To make these SDG-aware, we created an `SdgUserControl` class as follows.

1. We defined the class so it inherits from the standard `UserControl`, and declare that it implements the `ISdgMouseAndKeyWidget` (Figure 4, line 1). Inheriting the standard `UserControl` means it has all the methods, properties and event capabilities of a normal control (e.g., properties that define its location, extents, background and foreground colors, and font). It also means the programmer accesses this control through the .NET interface builder in the same way they access non-SDG controls.
2. The `SdgUserControl` then implements the SDG interfaces (lines 5-25). If the SDGManger finds this control under the current mouse coordinates, it invokes its SDG methods with the arguments filled in.
3. In turn, the `SdgUserControl` raises its own event corresponding to the received SDG event (lines 26-32). This new event is thus available to the end-programmer.

While this sounds complicated, this generic control was very easy to create given our design logic. For example, the complete class definition is handled in 32 lines of code. Figure 4 shows the complete code structure and how it handles two of the seven events.

```
public class SdgUserControl : UserControl,
                              ISdgMouseAndKeyWidget{
 public SdgUserControl() {
   // 3 routine lines of constructor code
 }

 // SDGManager invokes these methods when the mouse
 // moves over this control or when a keypress is
 // directed to the control. Note that each method
 // invokes the corresponding event handler
 public void OnSdgMouseMove(SdgMouseEventArgs e){
   if (SdgMouseMove != null) SdgMouseMove(this, e);
 }
 // The other 3 mouse methods are similar
 …

 public void OnSdgKeyDown(SdgKeyEventArgs e){
   if (SdgKeyPress != null) SdgKeyPress(this, e);
 }
 // The other 2 key methods are similar
 …

 // Now define the events
 public event SdgMouseEventHandler SdgMouseUp;
 public event SdgKeyEventHandler SdgKeyUp;
  // The other 5 events similar to the above
 …
}
```

Figure 4. The class definition of SDGUserControl

Example: Creating an Sdg ColorMixer Control

Using the `SdgUserControl`, programmers can now easily create their own SDG controls through techniques familiar to them. To illustrate this, we show how we can implement a trivial color-mixing control that fully responds to two mice (Figure 5, top). It is white if no one presses on the widget, blue if only the first person is pressing it, yellow if only the second person is pressing it, and green if both are pressing it at the same time. Figure 5 provides the complete code, omitting only the housekeeping code found in all .NET controls, and the two lines where we register the `SdgMouseDown` and `SdgMouseUp` event handlers. To explain its logic, the `press` array contains two elements, each holding the 'button press' state of the first and second mice. The `SdgMouseDown` event handler sets the appropriate `press` element to `true`, while the `SdgMouseUp` handler sets it to `false`. Both call the `Draw` method, which is a simple state machine that calculates which mouse or combination of mice are currently pressing the control, and sets the background color accordingly.

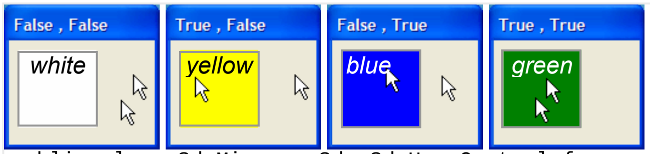
While simple, this example illustrates that the SDGToolkit makes SDG widget development straightforward.

EVALUATION: APPLICATIONS AND CONTROLS

The driving goal behind the toolkit is to let developers concentrate on the design of SDG applications rather than low level programming. This goal has been achieved in practice. While SDGToolkit is still fairly new, people are now using it to rapidly prototype single display groupware. This section illustrates a few early examples of what we and others have built.

Rush Hour: An SDG Game

Rush Hour is a simple online puzzle game, where the player must move cars around until they can get the special



```
public class SdgMixer : Sdg.SdgUserControl {
  private  Boolean [] press = new Boolean [2];
  // We omit initialization routines and
  // event handler registering as this is standard
  …
  private void SdgMixer_SdgMouseDown(
                   object s, Sdg.SdgMouseEventArgs e){
    press [e.ID] = true;
    Draw ();
  }

  private void SdgMixer_SdgMouseUp(
                   object s, Sdgt.SdgMouseEventArgs e){
    press [e.ID] = false;
    Draw ();
  }
  private void Draw () {
    this.Parent.Text = press[0].ToString () +
                   "," + press[1].ToString ();
    if ((press[0] || press [1]) == false)
      this.BackColor = Color.White;
    else if (press[0] && press [1])
      this.BackColor = Color.Green;
    else if (press[0]) this.BackColor=Color.Yellow;
    else this.BackColor = Color.Blue;
}}
```

Figure 5. The SDG ColorMixer control. *Colors are annotated.*

red car to the red exit marker (Figure 6). We decided to implement an SDG version of this game, where multiple players can move multiple cars simultaneously. First, we implemented a single user version of this game using the standard features of C# and .NET. Second, we modified this game to add multiple user capability via the SDGToolkit. This took less than one hour of straightforward programming (some extra programming was required to add collision detection for cars moving at the same time in the same position). The game is responsive and handles multiple players easily. We did not use our SDG Controls to implement the cars as we developed the game before the SDG Control layer was available.

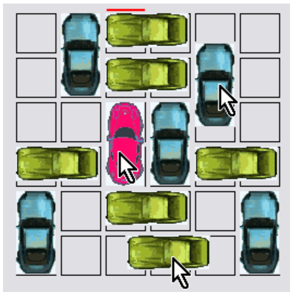


Figure 6. SDG Rush Hour

SDG Flow Menu – an SDG widget

To test our SDG widget layer, we recreated Guimbretière's Flow Menu [9] as an SDG interaction technique (flow menus use gesture as the primary interaction method, and are efficient for pen-based interfaces). Figure 7 shows the result, where each person has their own individual flow menu that can be raised any time (even concurrently) to select a pen color and pen size. The largest investment of time in developing this widget was on its non-SDG aspects, i.e., how to track and recognize a gesture. Making this SDG-aware was easy. First, because flow menus appear above the window (rather than within it) we could not use the `SdgUserControl` (which must live within the

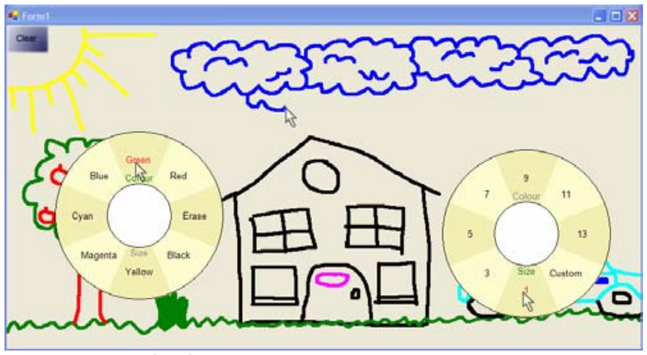


Figure 7. An SDG drawing application. Two users are select a drawing color and size from their individual flow menu, as the third is drawing.

window). Instead, the flow menu class implemented the mouse events defined in `ISdgMouseWidget` (the code is almost identical to Figure 4). Next, we used this within the drawing application by creating an instance of the flow menu for each mouse, and ensuring that the mouse down events reached the appropriate menu (about 3 lines of code).

SDG MagicLens ToolGlass

For our final example, Nicole Stavness (U. Saskatchewan) and Edward Tse recreated Bier's notion of a MagicLens ToolGlass [3]. Bier's ToolGlass was originally designed to exploit two handed input by a single user: one hand moved the ToolGlass over a surface (perhaps transforming how the underlying objects are displayed), while the other hand would 'click through' the ToolGlass to assign a property to the underlying object. For example, the ToolGlass could contain a palette of colors, and the user could position a particular color over and object and assign that color to it by clicking through it.

The SDG re-creation provides all users with their own magic lens (Figure 8). What is especially interesting about this is that each user has two pointing devices: one to move the lens (the cursor is the small hand in the bottom left corner of each Magic Lens) and one to click through (the arrow cursor). To our knowledge, this is the first ToolGlasses have been equipped with SDG semantics. This could easily be extended into a collaborative tool [5], e.g., where people can mix and select new colors by placing their lens atop of one another. As with the other examples, the programming effort to manage and identify multiple input devices was small compared to the effort in constructing the drawing application and the ToolGlass graphics.

RELATED WORK

MMM [2] was a wonderful early SDG breakthrough that illustrated concepts and challenges in SDG applications. It was built from scratch and required quite a bit of low level OS hacking to build a simple system that handled up to three mice. It was not a toolkit: to our knowledge no further work was done on it. Since then, many others have built proof of concept SDG applications through brute force.

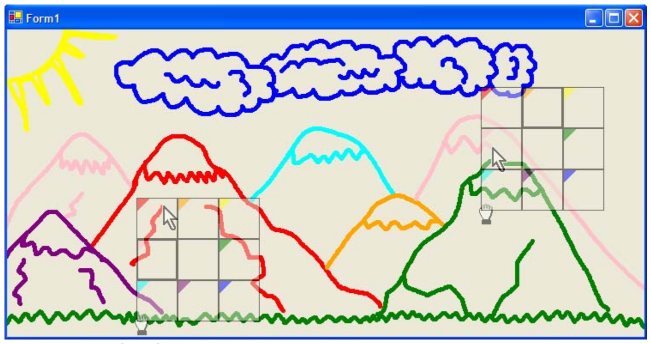


Figure 8. SDG MagicLenses. Each user moves his/her magic lens around with their non-dominant hand. With their other hand, they click through the lens to choose a color or the erase (middle square).

From a toolkit perspective, the most heavily commercialized work has been done in game console environments, as these come equipped with multiple input devices of various sorts, e.g., games controllers, steering wheels and foot pedals. However, they are not easy to develop on and consoles are not suitable for productivity applications.

Most operating systems do provide low-level facilities to acquire unusual input devices. In Windows, for example, the DirectInput SDK lets a programmer retrieve data from input devices not supported by the standard Windows API [http://msdn.microsoft.com]. These devices, however, are usually oriented toward gaming. While one could develop SDG applications on top of this API, it again would take considerable effort.

Pebbles [11] eschewed mice and keyboards and used multiple PDAs as input / output devices. Because PDAs are involved, it used a distributed model view controller paradigm to share data between the PDA and the computer running the SDG application (see also [7]).

Closest to SDGToolkit is MID [1], arguably the first generally released toolkit for SDG. Like SDGToolkit, it delivers multiple mice input as separate streams of events. To get these events, Java programmers coded classes that implement all of MIDs event handlers. MID has also been recently extended to work with other input devices, such as the DiamondTouch multi-touch display [4]. Otherwise MID is a subset of the SDGToolkit, where:

- it does not support multiple mice after Windows 98,
- it does not handle multiple keyboards,
- it only returns screen *vs.* window coordinates,
- it deals with the system mouse only by turning it off, which means that no conventional widgets are usable,
- it does not manage orientation issues in tabletop displays, and
- it does not provide any SDG widget building blocks[4].

[4] In spite of these limitations, the MID team constructed impressive SDG interaction techniques for children by combining it with the Jazz toolkit [5]. We also used MID for our earlier work on SDG [13]. MID obviously inspired our own development of SDGToolkit, and we are grateful to its creators.

CONCLUSIONS

SDG development parallels Gaines' [6] BRETAM phenomenological model of developments in science technology. The model states that technology begins with an insightful and creative breakthrough, followed by many (often painful) replications and variations of the idea. Empiricism occurs when people draw lessons from their experiences and formalize them as useful generalizations. This continues to theory, automation and maturity [6].

Within this context, the primary contribution of this paper is to move SDG technical work from the replication stage (where it is now) into the empiricism stage. We did this through several mechanisms. First, we articulated the technical requirements and challenges of SDG software that face many designers. Second, we detail solutions to these problems. We believe these are generalizable to most modern GUI windowing systems and that they can be used by other developers. Third, through our illustrations of how a programmer would develop an SDG application with our toolkit, we provide a conceptual model to other toolkit builders about how a toolkit for SDG should present itself. Finally, we provide the SDGToolkit itself as a resource that means others can work on the nuances of SDG and SDG interaction techniques rather than replicate SDG plumbing.

Our future plans follow several threads. First, we are now extending SDGToolkit's capabilities to manage other input technologies. These devices include display surfaces that recognize multiple touches such as the MERL DiamondTouch [4] and Smart Technology's DViT technology [www.smarttech.com]. We foresee no problem with this, as it merely means extending the way we now capture input and present events. Second, we are now using SDGToolkit to rapidly prototype and research many SDG applications and interaction techniques. In one project, we are creating software for linking distributed SDG settings (e.g., linking two or three SDG-enabled tables to one another). In another project, we are developing distortion-oriented information visualization techniques that give each SDG user a focus+context view into their information, centered around their cursor. In a third project with colleagues Sheelagh Carpendale and Russell Kruger, we are examining the social factors of how people use object orientation in SDG-enabled tables i.e., how they rotate artifacts to present them to others or to signal artifact 'ownership'. Finally, we are currently prototyping various types of SDG widgets such as the ones shown in Figure 7 and 8. These will be included in future versions of the toolkit as stock components. In all projects, the SDG toolkit is proving to be an extremely valuable resource.

If one looks down the road a few years, it is hard to imagine future computers that are not SDG-capable. This functionality could be achieved through an add-on such as SDGToolkit. At some point, our windowing systems should have SDG built into them as a fundamental component, and perhaps the concepts introduced in this paper will influence how this is done.

Acknowledgements. Michael Boyle and Tony Tang gave excellent technical assistance and feedback. Russell Kruger and Stacey Scott motivated our tabletop work. We are grateful to NSERC and Smart Technologies for funding.

Software availability. See www.cpsc.ucalgary.ca/grouplab/ for downloads, full documentation, and examples.

REFERENCES

1. Bederson, B. & Hourcade, J. Architecture and implementation of a Java package for Multiple Input Devices (MID). HCIL Technical Report No. 9908. http://www.cs.umd.edu.hcil. 1999.
2. Bier, B. & Freeman, S. MMM: A user interface architecture for shared editors on a single screen. *Proc ACM UIST'91*, 79-86, 1991.
3. Bier, E., Stone, M., Pier, K., Buxton, W. & DeRose, T. Toolglass and Magic Lenses: The See-Through Interface. *Proc SIGGRAPH '93*, 73-80, 1993.
4. Dietz, P. & Leigh, D. DiamondTouch: A multi-user touch technology. *Proc ACM UIST'01*, 219-266, 2001.
5. Druin, A., Stewart, J., Proft, D., Bederson, B. & Hollan, J. KidPad: a design collaboration between children, technologists, and educators, *Proc ACM CHI'97,* 463-470, 1997.
6. Gaines, B. Modeling and forecasting the information sciences. *Information Sciences* **57/58**, 3-22, 1991.
7. Greenberg, S., Boyle, M. & LaBerge, J. PDAs and Shared Public Displays: Making Personal Information Public, and Public Information Personal. *Personal Technologies* 3(1), 54-64, Elsevier, March 1999.
8. Greenberg, S. & Fitchett, C. Phidgets: Easy Development of Physical Interfaces through Physical Widgets. *Proc ACM UIST'01*, 209-218, 2001.
9. Guimbretiere, F. & Winograd, T. FlowMenu: Combining Command, Text, and Data Entry *Proc. ACM UIST'00*, 213-216, 2000.
10. Inkpen, K., McGrenere, J., Booth, K. & Klawe, M. The effect of turn-taking protocols on children's learning in mouse-driven collaborative environments, *Proc Graphics Interface*, 138-145, Morgan Kaufmann 1997.
11. Myers, B., Stiel, H., and Gargiulo, R. Collaborations using multiple PDAs connected to a PC.. In *Proc ACM CSCW'98*, 285-294, 1998.
12. Stewart, J., Bederson, B. and Druin, A. Single display groupware: a model for co-present collaboration, *Proc ACM CHI 1999*, 286-293, 1999.
13. Tse, E. and Greenberg, S. SDGToolkit: A Toolkit for Rapidly Prototyping Single Display Groupware. In *Extended Abstracts of ACM CSCW'02, 173-174,* 2002.
14. Zanella, A. and Greenberg, S. (2001) Reducing Interference in Single Display Groupware through Transparency. *Proc ECSCW'01*, Kluwer.