Building KidPad:
An Application for Children's Collaborative Storytelling

Juan Pablo Hourcade, Benjamin B. Bederson, Allison Druin

Human-Computer Interaction Laboratory

University of Maryland

3180 AV Williams Building

College Park, MD 20742

USA

(phone) +1 301 405 7445

(fax) +1 301 405 6707

jpablo@cs.umd.edu

Summary

Collaborating in small groups can be beneficial to children's learning and socializing. However, there is currently little computer support for children's collaborative activities. This was our motivation for building KidPad, a collaborative storytelling tool for children. KidPad provides children with drawing, typing, and hyperlinking capabilities in a large, two-dimensional canvas. It supports collaboration by accepting input from multiple mice. In building KidPad, we developed solutions to problems common to all single-display groupware applications for children: obtaining input from multiple devices, and using an intuitive user interface metaphor that can support collaboration. Our solution for obtaining input from multiple devices was MID, an architecture written in Java. We addressed the need for an appropriate user interface metaphor by using the local tools metaphor. This paper describes our work on MID and local tools in the context of building of KidPad, and aims to provide developers with valuable insights into how to develop collaborative applications for children.

Keywords

Children, collaboration, Java, single-display groupware, storytelling, zoomable user interfaces (ZUIs).

1. Introduction

Collaborative storytelling helps children develop interpersonal and story-related skills. Oral traditions are an example of how stories can provide an effective way of transferring and retaining information [Rubin 1995]. Storytelling also helps children develop communication skills. These skills are necessary for collaboration, and learning to work with others is another important skill to acquire [Wood 1996]. Collaborative storytelling is often present when children play. However, there is currently little computer support for children's collaborative storytelling. Computers can augment the collaborative storytelling experience by allowing for storage, and the ability to copy, share, and edit stories. They can also provide an excellent medium to create non-traditional forms such as non-linear stories.

These were the motivations for developing KidPad, a collaborative storytelling tool for children. KidPad provides drawing, typing and hyperlinking capabilities in a large two-dimensional zoomable space*. Through these capabilities, children can create stories by drawing story scenes and linking them together in two-dimensional space. KidPad supports multiple users through the use of multiple mice.

KidPad was created as part of KidStory, a three-year project (1998-2001) funded by the European Union's initiative for Experimental School Environments. KidStory was a collaboration between the Royal Institute of Technology in Stockholm, Sweden, the Swedish Institute of Computer Science, the University of Nottingham, and the University of Maryland. The aim of the KidStory project was to support children's collaborative learning experiences through technology. It accomplished this objective by forming an interdisciplinary, intergenerational, international design team that had the development of KidPad as one of its most important accomplishments.

KidPad's most distinguishing characteristic is its collaboration capabilities. We have consistently found in our research that children like to work with each other at a single computer [Druin et al. 1997] [Benford et al. 2000], and yet most software has no explicit support for this. We designed KidPad to support multiple simultaneous uses by giving each child control through their own mouse. The interface is designed explicitly to support this interaction style.

The collaboration model KidPad supports is called single-display groupware (SDG) [Stewart 1999]. In the SDG model, multiple devices, each controlled by a different user, provide input to one computer. In KidPad, each mouse controls a local tool on the screen, which acts as a cursor and holds the user's mode (e.g. a red crayon local tool draws in red color). Local tools are organized in toolboxes that can be opened (showing the tools), or closed (hiding the tools). Figure 1 shows KidPad with all its tools visible.

* KidPad is available freely for non-commercial use at http://www.kidpad.org.

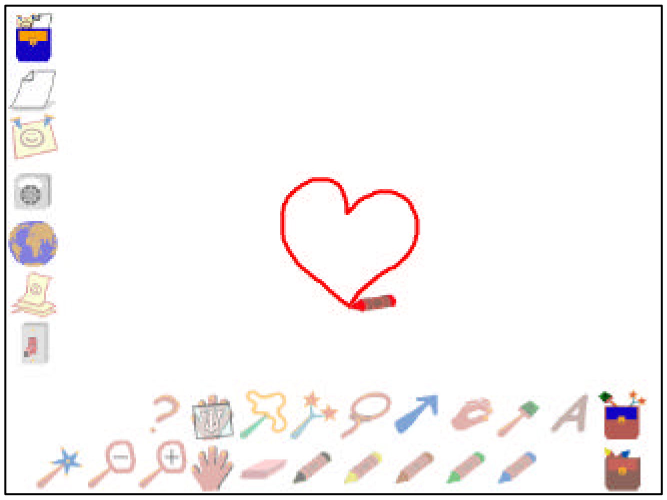

Figure 1: KidPad with all its tools visible

While KidPad appears to be a simple drawing application at first, it differs from most such applications in offering an enhanced authoring experience including zooming, panning, hyperlinking and collaboration capabilities. Children can create content in KidPad as they would in other drawing applications by drawing, typing and erasing. However, KidPad's use of vector graphics provides capabilities not usually found in bitmap graphics applications. For example, children can reshape their drawings and make them wiggle. KidPad also offers virtually infinite panning and zooming capabilities. Because of KidPad's use of vector graphics, drawings do not become pixelated when the view is zoomed in.

Links can be created from any drawing in KidPad to any other drawing or to a particular view of the KidPad canvas. When following a link to a drawing, the screen view animates so the drawing occupies the entire screen. When following a link to a view of the KidPad canvas, the screen view animates to the target view. KidPad's hyperlinking capabilities enable storytelling by allowing children to create links between story scenes they draw.

In order to build applications like KidPad, novel interaction mechanisms are required to support the unique needs of children. These in turn require software architectures, which this paper describes. In particular, we had to address two major issues that are common to all collaborative applications for children. The first was the lack of support for multiple input devices in existing operating systems and software development kits. The second was designing an appropriate user interface that would be intuitive for children and at the same time support simultaneous users on the same screen.

In this paper, we present a comprehensive solution to these issues. We introduce MID (Multiple Input Devices), a Java toolkit that supports applications receiving events from multiple devices of the same type; and we examine the evolution of the local tools metaphor for user interfaces.

2. Related Work

Other researchers have developed varied solutions for supporting single-display groupware applications. MMM was a simple SDG authoring tool developed at Xerox PARC in the early 1990's [Bier 1991]. It supported input from multiple mice and showed users' modes in "home areas". MMM used color to identify users and supported multiple users editing the same object at the same time. The architecture of MMM was not made available to the research community.

Bricker developed the Colt software kit as part of her Ph.D. thesis [Bricker 1998] [Bricker et al. 1998]. Her architecture could support input from multiple mouse-like devices. Using Colt, Bricker built many small applications for children that require users to collaborate. For example, in a chopstick game, users attempt to pick up beans while each controlling a chopstick. Bricker used color to differentiate users.

Stewart in collaboration with the second and third authors built the Local Tools architecture at the University of New Mexico to support the development of an early version of KidPad [Stewart 1999]. This architecture supported up to three input devices and only worked well with multiple mice. The architecture was based on Linux, XInput, a modified version of Tk, Pad++, and Perl. It was also tied to an early version of the local tools user interface metaphor.

The Pebbles project concentrated on the creation of SDG applications for personal digital assistants (PDAs) [Myers 1998]. The architecture behind the project tied user interface metaphors, those under the Amulet toolkit, to input from multiple PDAs. Amulet provided a set of features designed for building SDG user interfaces that include scrollbars, menus and palettes. PebblesDraw is a simple drawing application that was built using Pebbles. In it, objects can only be selected by one user at a time. PebblesDraw uses shapes to differentiate users and shows the users' current mode both in their cursor and in something similar to MMM's home areas.

ICON [Dragicevic 2001] is an editor designed to match input devices with actions on a graphical user interface. It is meant to bring the management of multiple input devices to a wider audience. ICON gives users the ability to connect input devices to input processors, which in turn connect to visual widgets. Programmers need to write an ICON module for each input device, input process and graphical widget to be used in ICON.

Klump was a children's collaborative character creation tool meant to aid in storytelling [Benford 2000]. The Swedish Institute of Computer Science, the Royal Institute of Technology (in Stockholm, Sweden), and the University of Nottingham developed Klump as part of the KidStory project. Klump supported up to two simultaneous users, taking input from two mice connected to separate computers but using only one display. Klump enabled children to create the characters of a story by molding a piece of play-dough-like material, a klump, on the screen. Children could squeeze and stretch the klump, apply face-like looks to it, change its colors, and rotate it. Klump encouraged collaboration by providing users with extra functionality if they both click on certain icons within a short timespan. It used colors to differentiate users.

3. MID (Multiple Input Devices)

A major challenge in writing SDG applications is getting input from multiple devices. MID (Multiple Input Devices) is a Java package developed at the University of Maryland that addresses this problem and offers an architecture to access advanced events through Java*. MID supports input from multiple devices of the same type and devices not supported by Java. In the following sections, we describe the features, architecture and limitations of MID.

MID provides developers using Java the ability to write SDG applications with a powerful yet straightforward and cross-platform way of getting input from multiple devices. MID is not tied to any user interface metaphor and was not designed to receive input from a particular type of device. However, we use MID in KidPad to support multiple simultaneous users through the use of multiple mice connected to the same computer, where each mouse controls one tool (such as a crayon, eraser, etc.).

We chose Java because of platform independence. MID consists of a cross-platform Java layer, and a native platform-specific layer. The cross-platform layer consists of an event manager and of event sources that have to be written for each device type (one event source per device type). A native class must be implemented on each platform for each device type for which data cannot be accessed through Java. Applications using MID use just the cross-platform Java layer, and therefore do not have to be changed for use on different platforms.

So far, device-specific event sources have been written for USB mice under Windows 98 and ME, mouse data over a socket, Radio Frequency ID (RFID) tag readers, and MERL's DiamondTouch [Dietz 2001].

3.1 MID Architecture

MID consists of an event manager that is a hub for all MID events, and event sources (one per device type) that generate events the event manager delivers to appropriate listeners. Figure 2 gives a visual overview of MID's structure.

* MID is available freely at http://www.cs.umd.edu/hcil/mid

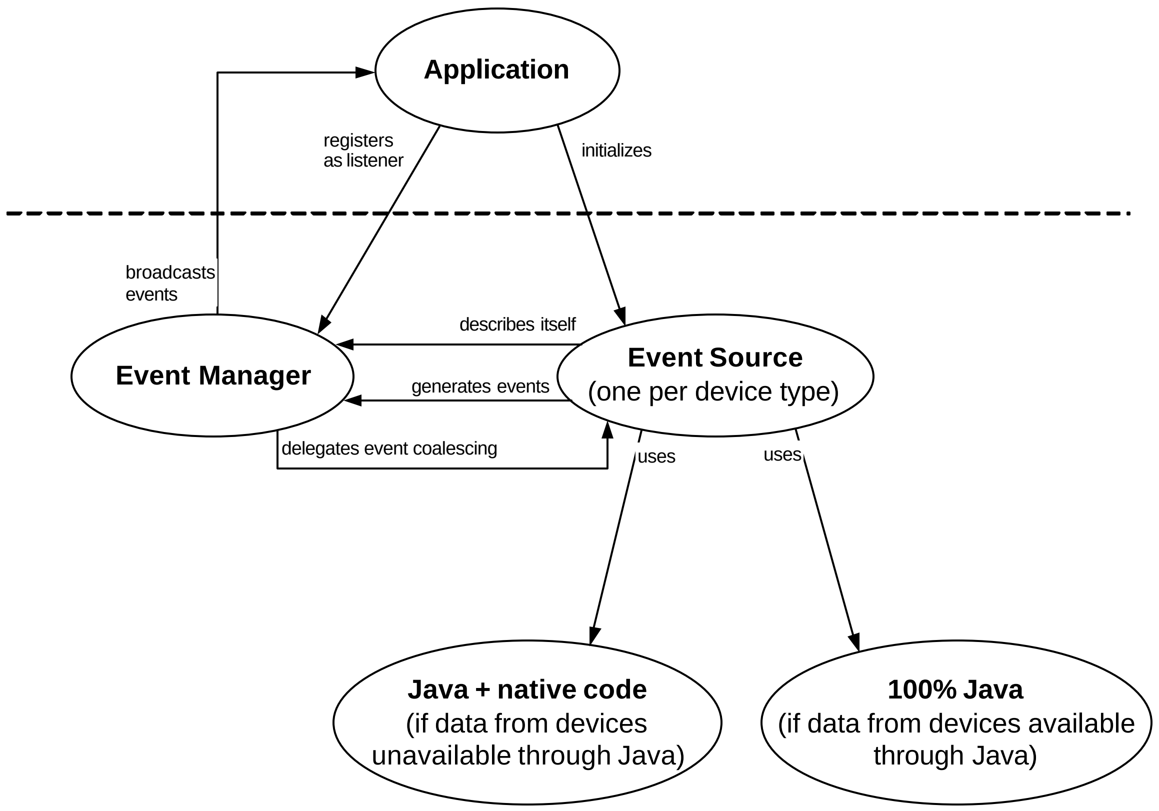
Application
registers as listener
initializes
broadcasts events
describes itself
Event Manager
Event Source (one per device type)
generates events
delegates event coalescing
uses
uses
Java + native code (if data from devices unavailable through Java)
100% Java (if data from devices available through Java)

Figure 2: Architecture of MID. The application initializes the device-specific event sources and registers to listen to events with the event manager. The event source uses a combination of Java and native code if data from the devices is unavailable through Java and 100% Java code otherwise. The event source describes itself to the event manager and tells it when events should be generated. The event manager broadcasts the events to the application. It also delegates event coalescing decisions to the event source.

3.1.1 Event Manager

The event manager handles the registration of listeners and the broadcasting of events to these listeners. It receives events from event sources. The event manager has no knowledge of any of the device specific event sources and will therefore work with any event source written in the future. It is designed to make it as easy as possible for developers to add support for types of devices Java doesn't support.

Event sources have to specify to the event manager what type of events they support, what listener interfaces these events may be sent to, and what devices they have available. Each device may generate multiple types of events, and there may be multiple devices for each device type. Each event type may be sent to multiple listener interfaces.

When the event manager receives an event from an event source, it first posts it to the Java event queue. When the Java event queue gives the event manager the possibility of coalescing the event, the event manager delegates this task to the event source that generated the event. When the event is ready to be broadcast, the event manager sends it to all the appropriate listeners.

MID's event manager also supports the dynamic addition and removal of input devices by being a source of device change events. Objects that register to listen to this type of event are notified when a device is added or removed.

3.1.2 Event Sources

Event sources are responsible for providing all the needed device-specific information for the type of device they support. Their basic responsibility is to know when events should be generated. They may also provide utility methods related to the particular type of the device they support. Event sources work together with listener interfaces and event classes written for the particular type of device the event source deals with.

Event sources communicate with native code in order to know when to generate events only if they cannot get data from the devices through Java. We have studied two different ways of communicating with native code. One way is to have the native code in a library and use the Java Native Interface to communicate with it. The other way is to use sockets for communication with an event server written in a separate process (which could be Java or native code, and could even come from another machine). When event sources generate events they send them to the event manager, which is responsible for delivering them to the appropriate listeners.

Event sources are responsible of informing the event manager of the event types and listener interfaces they support as well as of the devices they have available. They are also responsible of notifying the event manager when devices are added or removed. Event sources have the responsibility of deciding when to coalesce events.

If it makes sense for the type of devices they support, we recommend that event sources revert to some kind of default behavior if these devices are unavailable. This behavior should be coded solely in Java. Applications that follow this recommendation will work in some limited way when data from the devices is not available.

3.1.3 MID Mouse Event Source

The MID mouse event source, which is used in KidPad, was the first event source implemented for MID. It generates events from USB mice under Windows 98 and ME. It reverts to Java mouse events if multiple mice are not available. This ensures that applications that use this event source will always work, although possibly with only one mouse.

The MID mouse event source communicates with a native code library through the Java Native Interface in order to know when to generate events. The native code uses the Microsoft DirectInput API to get input from USB mice.

The MID mouse event source allows developers to set the location of mice, apply motion constraints, and the time span used for counting mouse clicks. Setting the location of mice is particularly useful for an application that has to define its own cursors and needs to give them an appropriate initial location on the screen. Constraining the motion of all mice and/or the motion of particular mice can be used to keep the cursor within the visible area of the window. Another application would be to divide the screen into regions and have only one mouse operate in each region.

Another important feature of the MID mouse event source is that it enables applications to access all of the input buttons and movement axes on the particular mouse in use. Thus, a mouse with a wheel generates a third button event and motion on the z-axis.

When supporting a mouse event's access to the keyboard's control, shift, and alt modifier keys, this event source currently assumes that there is only one keyboard available. This access to the keyboard through mouse events demonstrates the low level at which the assumption of a single mouse and keyboard has been made. When MID supports multiple keyboards, we will have to handle this differently.

The implementation of event coalescing in this event source coalesces move and drag events if they come from the same mouse. This means that events can be combined if too many pile up on the event queue.

3.1.4 MID Socket Mouse Event Source

The MID socket mouse event source provides support for mouse events received over a TCP/IP socket connection. These events are generated by a program running in a process separate from MID and are sent across a socket. The MID socket mouse event source receives the events, interprets them, and generates corresponding MID mouse events that are passed on to the MID event manager. The default data this source expects to receive through the socket is Java mouse events. However, this event source may be subclassed to provide other interpretations of the data received through the socket.

3.1.5 MID Switch Event Source

The MID switch event source, created by our KidStory colleagues at the Swedish Institute of Computer Science, generates switch events for Radio Frequency ID (RFID) Tag readers. The MID switch event source generates switch events for the tag reader by monitoring the serial port the reader is connected to. Each time a tag is introduced into the reading field of the reader a switch closed event is sent to the MID event manager. When a tag ceases to be detected through the serial port, a switch opened event is generated. We have used this event source in an augmented version of KidPad designed to work in a room environment.

3.1.6 MID DiamondTouch Event Source

DiamondTouch [Dietz 2001] is an input device developed at Mitsubishi Electric Research Laboratories that supports multiple simultaneous users interacting with it. It provides a touch-surface that can be used in combination with a tabletop display. The technological advantage of DiamondTouch is that it is able to recognize multiple simultaneous users as they touch its surface. This is a great improvement over what could be accomplished by a large touchscreen.

Working with an early beta version of a DiamondTouch, we wrote a MID event source that generates MID mouse events. We easily integrated this event source with KidPad and had a few successful sessions with children where they got to interact with KidPad through a DiamondTouch.

3.2 MID Versus Typical Java Event Sources

MID follows the Java event model, having event objects sent from event sources to event listeners. Listener interfaces and event classes written for MID are comparable to those written for typical Java event sources. However, MID event sources are simpler to implement than typical Java event sources. On the other hand, the fact that MID supports multiple devices has the side effect of requiring listeners to register to listen in a slightly different way than they would with typical Java event sources.

3.2.1 Differences in event source implementation

MID event sources have to inform the MID event manager of the event types and listener interfaces they support as well as the devices they have available.

Standard Java event sources have to manage the registration of listeners and the broadcasting of events; something MID event sources don't need to do. This requires the implementation of multicasters. Standard Java event sources that deal with only one device can manage with one multicaster. When dealing with multiple devices, the task becomes more complicated as separate multicasters have to be kept for each device, with an extra multicaster for listeners that want to listen to all devices. Standard Java event sources also need to implement methods listeners use to register to listen.

```
// Java code using standard Java events to access mouse and mouse motion events
//
class MyClass implements MouseListener, MouseMotionListener {
        // Constructor adds mouse and mouse motion listeners
    public MyClass(Component myComponent) {
        . . .
        myComponent.addMouseListener(this);
        myComponent.addMouseMotionListener(this);
        . . .
    }

        // Mouse listener events
    public void mousePressed(MouseEvent e) {
        . . .
    }
    public void mouseReleased(MouseEvent e) {
        . . .
    }
    . . .

        // Mouse motion listener events
    public void mouseMoved(MouseEvent e) {
        . . .
    }
    public void mouseDragged(MouseEvent e) {
        . . .
    }
}
```

```
// Java code using MID events to access mouse and mouse motion events
// from multiple USB mice.
//
class MyClass implements MIDMouseListener, MIDMouseMotionListener {
    int numberOfDevices;  // Number of devices available through MID
    public MyClass(Component myComponent) {
        . . .
       MIDMouseEventSource midSource = MIDMouseEventSource.getInstance(myComponent);
       MIDEventManager midManager = MIDEventManager.getInstance();
       numberOfDevices = midManager.getNumberOfDevices();
       midManager.addListener(MIDMouseListener.class, this);
       midManager.addListener(MIDMouseMotionListener.class, this);
        . . .
    }

        // Mouse listener events
    public void mousePressed(MIDMouseEvent e) {
       int device = e.getDeviceID();       // Use mouse ID in application-specific manner
        . . .
    }
    public void mouseReleased(MIDMouseEvent e) {
       int device = e.getDeviceID();       // Use mouse ID in application-specific manner
        . . .
    }
    . . .

        // Mouse motion listener events
    public void mouseMoved(MIDMouseEvent e) {
       int device = e.getDeviceID();       // Use mouse ID in application-specific manner
        . . .
    }
    public void mouseDragged(MIDMouseEvent e) {
       int device = e.getDeviceID();       // Use mouse ID in application-specific manner
        . . .
    }
```

Figure 3: The top code fragment shows Java code that gets standard Java mouse and mouse motion events. The bottom code fragment shows Java code using MID events to access multiple mice.

3.2.2 Differences for listeners

While reducing the amount of coding required for event sources, MID also provides users with a consistent way of accessing events from multiple devices. This has the cost of making registering to listen to events a bit different than it is for standard Java event sources.

As an example, we will examine the differences between using Java mouse events and MID mouse events. A class that receives Java mouse events has to register with a component (usually the visible window where the events will occur) to get those events.

The class also has to implement the methods specified in the MouseListener and MouseMotionListener interfaces. These methods will be called when mouse events occur, and a MouseEvent object describing the event will be passed to them. The top of Figure 2 shows sample code of a class that uses Java mouse events.

The bottom of Figure 2 shows sample code of a class that uses MID mouse events. The most noticeable difference between this code and the code that uses Java mouse events is two lines of code that get a MID mouse event source (MIDMouseEventSource) object and a MID event manager (MIDEventManager) object.

Instead of registering with a component, the class has to register with the MID event manager to listen to MID events. The method call used for registering to listen to events is the same for any kind of listener because the listener class is passed as a parameter.

The MID event manager offers the flexibility of registering to listen to all devices or to a particular device. This supports applications written in two styles. The first style dispatches events from all devices to each listener. It is up to the listener to determine which device generated the event by calling a method on the event object that returns the ID of the device that generated the event.

An alternative application style is to register to receive events from a specific device. Then, an application would write one listener for each device, which would support decoupling of the devices.

MID also allows listeners to register to listen to events from a particular event source. This is useful for applications that want to react differently to events of the same type depending on what event source generated them. In these cases, applications can implement one listener per event source to support decoupling of the event sources.

In addition, there is a call to the MID event manager that returns the number of devices currently available. This turns out to be necessary for most applications so they can build internal data structures and mechanisms to support the available input devices. When more than one type of device is available through MID, the application has to loop through the available devices to build the appropriate data structures. This can be easily accomplished through the MID manager.

The MID mouse event source object is instantiated so that it may start sending events to the MID event manager (in case it hadn't previously been initialized). It can also be used to provide functionality specific to the type of device it deals with, as described in the section dedicated to the MID mouse event source.

The fact that event listeners receive events from the MID event manager instead of event sources can make it simpler to test event listeners. For example, an event listener that in production would receive input from an event source that has a complicated physical setup could be debugged using an event source with a simpler setup that generates the same type of events. If the event source with the complicated physical setup reverted to a simpler setup solution when its original setup was not available, then the event listener code would not have to be changed. If this were not the case, the only change needed would be to initialize the simpler setup event source instead of the one with the complicated physical setup. Without MID, the event sources would likely have different APIs, Sforcing developers to write more custom code for testing purposes.

3.3 Support for Multiple Output Devices

MID may also be used to send output to devices. An application wishing to send output to device A would act as an event source and the code that can actually send output to device A would act as a listener of these events. This way, an application can be shielded from the details of the implementation of sending output to a particular device. MID would also provide the advantages of an event queue and event coalescing.

3.4 Layers of MID

An application may also use MID in layers in order to construct higher level events from lowe r level ones. When using MID in layers, the bottom layer is only an event source and the top layer is only a listener of events. All the layers in the middle are both event sources and listeners of events. These middle layers listen to events from the layer below and create events that are listened to by the layer above.

Event layering can be used to generate higher-level events. For instance, a gesture recognizer could listen to mouse events and be a source of gesture events.

We tested this concept by building a simple demonstration application that consisted of three layers. The bottom layer was the standard MID mouse event source, and generated events from several mice. A middle layer listened to those mouse events and used the mode of the left button on each mouse to specify bits of a number. Thus, four mice could generate the numbers 0 through 15. The middle layer was also a source of "number" events that generated an event with the computed number whenever a mouse button was pressed or released. Finally, a top layer listened to the number event and printed it out.

Although this application could have been implemented by having each event layer listen directly to the other event layer, using MID provided the advantage of a level of indirection. This way, each listener of events only has to know that there is a source of events somewhere, but does not have to know where it is, or what code is generating those events.

So, in the test application just described, the fact that the "number" events came from a specific module that computed its numbers from mouse buttons was completely unknown to the top layer. It simply listened to "number" events. For testing purposes, a different "number" generator could have used a keyboard. We have often heard programmers say "when in doubt, add a level of indirection". This is typically meant half in jest, but half seriously as well. Adding a level of indirection is a powerful software engineering principle that decouples related modules. MID provides a stru cture for adding a level of indirection in event-driven multi-layer programs.

3.5 Extending MID

MID can be extended in two ways. One way is to add support for devices already supported by MID through native libraries on platforms not currently supported by MID. Adding such support consists of creating a native library in the unsupported platform that interfaces correctly with the existing event source.

MID can also be extended by adding support for new devices. Adding support for such devices consists of adding device specific event sources for each new type of device. These event sources would use native code in order to know when to generate events if data from the devices cannot be accessed through Java. Event classes and listener interfaces would also have to be written.

3.6 Limitations

The most limiting aspect of MID is that it currently has only four implemented event sources. These sources themselves have their limitations.

The MID mouse event source works only with USB mice under Windows 98 and ME. The mice have to be USB mice because of a limitation of Windows. Windows merges the input from the non-USB mouse with the input from USB mice into one event stream. Windows 2000 merges input from all mice at a very low level, making separate streams inaccessible to MID. Windows XP appears to have a new API that may give access to multiple streams of mouse data at a high level, but we have not yet tried to use it for MID.

Another limitation of the MID mouse event source is that it takes over the system cursor and users cannot send mouse input to other windows besides the application's window unless they switch to them through the keyboard. This was done on purpose because the alternative is to have both mice control the single system cursor.

In the switch event source, the current software in the RFID tag reader only outputs data on the serial line when it is powered up or a tag is placed on it, thus there is no way to support a plug and play behavior.

Another limitation of the switch event source is that it supports only one reader. This fact makes the switch event source an example of MID being used to support input from devices that Java does not support rather than an example of MID being used to support input from multiple devices.

3.7 Performance

While we have not measured the time to generate MID events, we have compared it to standard Java event code, and there are no noticeable differences that affected performance. In addition, we have tested MID by using four mice simultaneously with KidPad running on a 266 MHz Pentium II laptop, and performance is acceptable (i.e. cursors followed mouse movement smoothly).

4. KidPad's Interface

While MID solves the problem of obtaining input from multiple devices, developers must design appropriate software user interfaces in order to effectively support children in collaboration. The following sections summarize how we designed KidPad's user interface.

4.1 Interface Design Process

In designing KidPad, we followed the Cooperative Inquiry approach described in [Druin 1999]. Our group was an interdisciplinary, intergenerational and international team. It was made up of computer scientists, educators, psychologists, artists, teachers and children. We worked closely with three groups of children, aged 7 to 11, as design partners: one small group at the Human-Computer Interaction Lab at the University of Maryland and two in elementary schools in Sweden and England.

We started the design process by working with a version of KidPad whose user interface was based on an earlier version of KidPad, built at the University of New Mexico [Druin 1997]. We worked in parallel with our child design partners at the three locations and went through several design iterations, including six major releases. Sessions of work with our child design partners provided feedback that guided us in modifying KidPad. Our design partners were always happy to see changes they suggested in the next release of KidPad. The design iterations brought many new tools to KidPad and also substantially modified the concept of local tools we used for our interface.

4.2 Local Tools

We designed KidPad to use "local tools" instead of menus or tool palettes [Bederson 1996]. Local tools are graphical objects that act as cursors and hold their own mode (e.g. a red crayon local tool draws in red at its current location). Local tools provide a concrete visual interface that is very easy to learn for young children. In addition, this metaphor works well with the ability to work with multiple mice at the same time [Stewart 1999]. Multiple local tools can be active simultaneously, where each user controls one local tool at a time. Users can change the local tool they are using by picking up an unused local tool or by exchanging local tools with another user. Some local tools change their behavior when used collaboratively [Benford 2000]. For example, two crayons that start drawing near each other combine colors and draw a filled shape.

The goal of local tools is to support co-present collaboration by avoiding problems with multiple users regarding modes. Traditional alternatives such as menus and palettes assume that there is one global mode in the application. For example, drawing programs will assume there is a global ink color and pen thickness. While this assumption works well when there is only one user, it is no longer appropriate when dealing with multiple users, as each user needs a mode of his/her own. By holding their own mode, local tools get around this problem.

4.2.1 Design Evolution

As mentioned earlier, through our work with our child design partners we have been able to improve the usability of the local tools metaphor. The following is a discussion of the issues we have had to deal with and our solutions to them.

Initially, local tools lay anywhere on a large two-dimensional space. While active tools were always visible, inactive tools could be invisible if the user(s) had panned or zoomed since the tool had been dropped. A click on a special icon could be used to bring the tools back to the visible area of the working canvas. The fact that tools could be invisible to the user violated the principle of recognition over recall, as the user had to remember that there were other tools available. It also meant that users often had to take an extra step to use a tool.

We first solved this problem by having the tools remain visible all the time, but still stay where they were dropped. Instead of moving with the authored content in the two dimensional space, they remained at the same screen location (what in zoomable user interface jargon is known as "sticky").

This approach worked well until we added a few new tools to KidPad. Soon, the tools were getting in the way of authoring content, obstructing portions of the screen. They were also difficult to find, as they would end up located in different parts of the screen, forcing users to scan the entire screen in order to find a particular tool.

To deal with this problem, we came up with the idea of using toolboxes. Toolboxes, represented by icons, hold tools and can be open or closed. If a toolbox is open, its tools are visible. If it is closed, its tools are not visible. When a user opens a toolbox, its tools appear by its side. Closing a toolbox removes its tools from the screen, giving more space for authoring content. Toolboxes can therefore be used to quickly organize tools and better manage the authoring space.

Originally, users could change the tool they were holding by either picking up another tool or dropping the tool they were holding. Each user had a hand tool (used for panning and following links) that was picked up whenever a tool was dropped. Having one hand tool per user contributed to clutter on the screen. Having two different types of actions to switch tools also made the interface complicated, especially for children who were the intended users of the application.

We reached a solution to these problems by having only one method of switching tools: picking up another tool. We therefore avoided the need to drop a tool, getting rid of an unnecessary function. By eliminating the drop function we were also able to eliminate all the hand tools except for one (i.e. hand tools became no different from the other tools). Since the hand tools were the tools held by users when KidPad was started, we had to choose other tools for users to hold when starting KidPad. Our choice was to pick up crayons. This choice has shown promise as it has encouraged children to start drawing and experimenting with authoring right away.

While the solutions we implemented helped in reducing clutter on the screen, we still had clutter problems because when tools were picked up, dropped tools ended up in various parts of the screen, obstructing the authoring area. To solve this problem, we decided that a dropped tool should go back to its original location next to its toolbox. If its toolbox is closed, then it goes inside the toolbox and becomes invisible. Therefore, tools only take up space next to the toolboxes, which can also be closed. This change reduced clutter.

As we have been adding tools to KidPad, the space that the tools take up next to the toolboxes has become a bit of a concern. We have implemented the simplest solution to this problem: allow for higher resolutions. We were using an 800x600 pixel default resolution, while most recent computer systems can easily handle higher resolutions. We have therefore added options to run KidPad at higher resolutions.

We have also recently rearranged the tools and toolboxes. While all toolboxes used to be at the bottom of the screen, we moved one to the top left of the screen. This toolbox holds tools that cannot be picked up and therefore do not behave like normal local tools. They handle loading, saving, printing and exiting (similar to a file menu). While we originally had only crayon tools and an eraser tool in the bottommost toolbox (which is open by default), we moved the panning, zooming and linking tools to it, to encourage children to experiment with these tools.

Through our work with children we have found that they often have trouble collaborating with each other [Benford 2000]. Therefore, we developed a way to encourage collaboration by providing some tools with collaborative behaviors. If two tools of the same kind are used together, they behave in a different manner. For example, two crayons fill the area between them with their combined color. We also added collaborative behaviors to a few other tools [Benford 2000].

These collaborative behaviors brought on the need for having more than one tool of the same kind. In order to accomplish this, we added a cloning tool that copies tools, creating a "clone". We later extended the capabilities of the cloning tool to copy authored material.

4.2.2 Trade-offs of Local Tools

The main advantage of local tools is that they easily deal with the issue of modes in a multi-user environment. Having each tool hold the user's mode takes care of this. The other important advantage of local tools is that they make sense to children as using them resembles the way they use tools such as pencils and erasers in their everyday activities. In this way, local tools are less abstract than menus or palettes (users don't have to look elsewhere to see what their mode is), and thus are easy to learn.

The main problem we have found with local tools is the amount of space they occupy. The evolution of local tools has had to deal primarily with how to make them more space efficient. Even given our latest changes, the main issue with local tools is that they do not scale very well. In this aspect, they face the same problem palettes face and are only useful up to a moderate number of tools.

A possible solution to this problem is to have local tools behave a bit more like menus and less like palettes. The toolboxes would be closed by default. When clicked on, they would open, and the user would be able to pick up a tool. After the tool is picked up, the toolbox would close again. The advantage of this method is that space would always be available. The disadvantage is that it would sometimes take an extra click of the mouse to get to tools, and users would have to remember in what toolbox the tool they want resides. It may be that just like WIMP (windows, icons, menus, and palettes) applications use a combination of menus and palettes, SDG applications can use a combination of local tools approaches.

4.3 Toolbox Design

KidPad's tools are currently organized in three toolboxes. Two toolboxes are located on the bottom right of the screen and their tools appear to their left when they are open. A third toolbox is on the top left of the screen and its tools appear below it when it is open.

When KidPad is started, the bottommost toolbox is open while the rest are closed. The bottommost toolbox (Figure 4) holds the basic tools needed to tell stories in KidPad. It holds crayons to draw; an eraser to fix mistakes; a hand tool to pan the drawing area and follow hyperlinks; zoom-in and zoom-out tools; and a magic wand to create hyperlinks between story scenes.

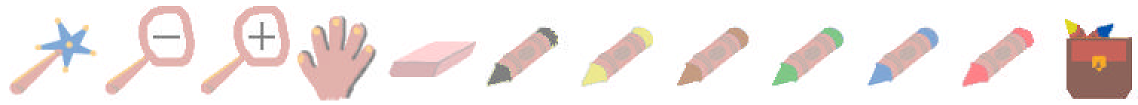

Figure 4: Bottommost toolbox

The other toolbox at the bottom of the screen (Figure 5) holds tools that help better manage stories and add some special features. The selection tool is used to select and move objects. The grouping tool makes pieces of a drawing act as one. The text tool types text. The turn-alive tool animates shapes, making them wiggle. The clone tool makes copies of tools and objects. The filler is used to fill line shapes with color. The puller tool is used for reshaping lines and polygons. The x-ray tool makes x-ray windows. Drawing inside an x-ray window makes that drawing visible only through that x-ray window. The help tool plays audio files describing the functionality of other tools when moved over them. When picking up another tool, it demonstrates that tool's functionality.

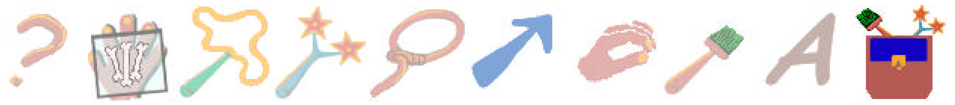

Figure 5: Second toolbox from bottom

The toolbox on the top left of the screen holds tools that cannot be picked up. These handle some of the items commonly found in "File" menus in many applications. They support starting a new story, loading stories, saving the current story, saving the current story in HTML format, printing the current story, and exiting the application. When one of these tools is clicked on, the operation is carried out, and the user keeps the tool he/she had before clicking.

The tool for loading stories takes the user to another screen (Figure 6) where he/she can pick a previously saved story from a bulletin board. Clicking on a story's thumbnail loads the story, while clicking on the thumbnail representing the current story (at the bottom of the screen), takes the user back to the current story. This screen used to also include icons for saving, saving as HTML, printing and exiting. Through children's feedback we found that the earlier interface was confusing and therefore opted to remove the functionality unrelated to loading stories. Those options are now available as tools in the toolbox on the top left corner of the screen.

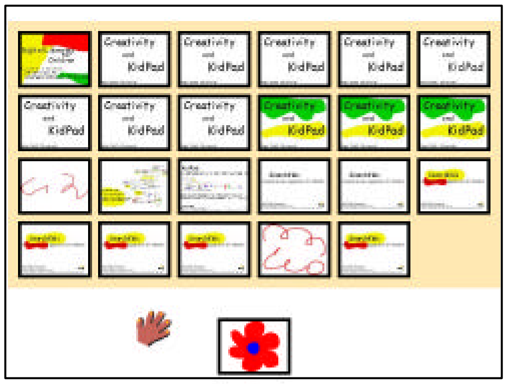

Figure 6: Bulletin Board

4.4 Lessons Learned about Children and Mouse Interaction

Through observing children who had not previously used KidPad we have learned that local tools are an intuitive interface. Most children figure out on their own how to pick up and operate the tools in very little time.

Through our work with children we have developed some guidelines to follow when designing mouse-driven software for young children. When clicking, children cannot reliably distinguish between single and double-clicking. In many occasions, they click several times. Therefore, we stayed away from providing different functionality when single and double-clicking. Instead we used single-clicks for interactions. After a click is processed, all other clicks in the event queue are discarded.

Just as children cannot be relied on to count the number of times to click, they cannot be relied on to click on a particular mouse button. From our experience, many young children cannot distinguish between the left and right mouse buttons [Druin et al. (1997); Druin et al. (2001); Hourcade et al. (2002a); Hourcade et al. (2002b)]. Our experience is supported by research showing that children do not achieve orientation with respect to themselves until age 6, and cannot apply the concepts of left and right relative to other objects until age 8 [Kaluger & Kaluger (1979)]. We therefore assigned the same functionality to all mouse buttons.

Children also have difficulty using software with small visual targets. This is due to children's developing motor skills. Fitts' law studies have shown that young children's performance in pointing tasks is significantly lower than that of adults [Kerr (1975); Salmoni & McIlwain (1979); Wallace, Newell & Wade (1978)], suggesting that they require larger visual targets in graphical user interfaces. Our observations of young children using graphical user interfaces support this suggestion.

4.5 Observations of KidPad's Collaborative Use

One of our goals in the design of KidPad is to support collaboration in the creation of stories. Early on, we decided that in order to effectively support collaboration, all parties involved in the collaboration had to have equal access to the application. In order to support this vision we had to look for hardware and software solutions. We chose to support multiple mice on the hardware side. We also considered tablets, but no USB tablets were available when our project started, making support for multiple tablets more difficult. Mice were also attractive because of their low cost and wide availability. On the software side, we implemented local tools, described in a previous section.

In spite of all the support for collaboration we built into KidPad, we still found that if children do not want to work together, they will not collaborate. They will very often decide to work independently, or even worse, compete against each other with scribble wars, or erasing each other's creations.

As mentioned earlier, in order to encourage collaboration, we added special collaborative behaviors to some of the tools. When using these tools together, users can access extra functionality not available when working independently. Of the tools we added collaborative behaviors to, crayons were by far the most popular in collaborative mode. This might be due to the fact that they are the type of tool children are most likely to be holding at the same time. Through our informal observations we found that these collaborative behaviors encouraged communication between users and helped spark collaboration. They were no silver bullet though, as children who did not feel like working with each other still did not collaborate.

While our goal has been for children to collaborate while using KidPad, we have found that KidPad supports multi-generational collaboration. We found this out by chance, as one of our research institutions was hosting an open house. We setup KidPad on several computers and advertised our lab as a place where there were activities for children. While our original intention was to have children work with each other, many children started using KidPad with their parents. We were pleasantly surprised to find out how well they collaborated. None of the visitors had seen KidPad before, yet they managed to use it and collaborate without trouble. These observations are a strong suggestion that KidPad may be a good application to bring the family together at homes. The other lesson we learned from this experience is that social roles are very important when it comes to collaboration. While the parent-child role worked very well, other roles between children may lead to competition rather than collaboration.

5. Conclusion

In this paper we have presented our experience in building KidPad. We believe the combination of an architecture like MID, and the use of local tools can provide future developers of single-display groupware applications for children powerful building blocks. MID provides a valid example of an architecture to support input from multiple devices and may be used for single-display groupware applications in general. Local tools provide a solution to the problem of global modes and are easy to understand for children.

6. Acknowledgements

We would like to thank the European Union's i3 initiative for funding the development of KidPad through the KidStory project, and our KidStory colleagues Angela Boltman at the University of Maryland, Camilla Abnett, Victor Bayon, Steve Benford, Sue Cobb, Rob Ingram, Helen Neale, Claire O'Malley, Danae Stanton and John Wilson at the University of Nottingham, Carina Fast, Marita Kjellin, Yngve Sundblad and Gustav Taxén at the Royal Institute of Technology in Stockholm, Sweden, and Karl-Petter Åkesson, Pär Hansson, Jan Humble, Kristian Simsarian and Jonas Söderberg at the Swedish Institute of Computer Science (SICS). We also thank our child design partners in Maryland, England and Sweden. In addition, we like to acknowledge Jason Stewart and his colleagues at the University of New Mexico who worked on an earlier version of KidPad.

Bibliography

1. Bederson, B. B., Hollan, J. D., Druin, A., Stewart, J., Rogers, D., & Proft, D. (1996). Local Tools: An Alternative to Tool Palettes. *In Proceedings of User Interface and Software Technology (UIST 96)* ACM Press, pp. 169-170.

2. Bederson, B. B., Hollan, J. D., Perlin, K., Meyer, J., Bacon, D., & Furnas, G. W. (1996). Pad++: A Zoomable Graphical Sketchpad for Exploring Alternate Interface Physics. *Journal of Visual Languages and Computing, 7*, 3-31.

3. Bederson, B. B., Meyer, J., & Good, L. (2000). Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java. *In Proceedings of User Interface and Software Technology (UIST 2000)* ACM Press, pp. 171-180.

4. Benford, S., Bederson, B. B., Akesson, K., Bayon, V., Druin, A., Hansson, P., Hourcade, J. P., Ingram, R., Neale, H., O'Malley, C., Simsarian, K., Stanton, D., Sundblad, Y., & Taxén, G.

(2000). Designing Storytelling Technologies to Encourage Collaboration Between Young Children. *In Proceedings of Human Factors in Computing Systems (CHI 2000)* ACM Press, pp. 556-563.

5. Bier, E. A., and Freeman, S. (1991). MMM: A User Interface Architecture for Shared Editors on a Single Screen. *Proceedings of User Interface and Software Technology (UIST 91)*, ACM Press, pp. 79-86.

6. Bier, E. A., Freeman, S., and Pier, K. (1992). MMM: The Multi-Device Multi-User Multi-Editor. *Proceedings of CHI '92:Human Factors in Computing Systems*, ACM Press, pp. 645-646.

7. Boltman, A., Druin, A., Bederson, B., Hourcade, J.P., Stanton, D., Fast, C., Kjellin, M., O'Malley, C., Cobb, S., Sundblad, Y., Benford, S. (2002). The Nature of Children's Storytelling With and Without Technology. Submitted to American Educational Research Association (AERA) Conference 2002 (submitted).

8. Bricker, L. J. (1998). *Collaboratively Controlled Objects in Support of Collaboration*. Doctoral dissertation, University of Washington, Seattle, Washington.

9. Bricker, L. J., Baker, M., Fujioka, E., Tanimoto, S. (1998). *Colt: A System for Developing Software that Supports Synchronous Collaborative Activities*. University of Washington Technical Report (UW-CSE-98-09-03).

10. Dietz, P., Leigh, D. (2001). Tactile user interface: DiamondTouch. In *Proceedings of User interface software and technology (UIST 2001)*, ACM Press, pp. 219-226.

11. Dragicevic, P, Fekete, J.D. (2001). Input Device Selection and Interaction Configuration with ICON. *Actes de la Conférence Internationale IHM-HCI 2001*, Blandford, A.; Vanderdonckt, J.; Gray, P., (Eds.): People and Computers XV - Interaction without Frontiers, Lille, France, Springer Verlag, pp. 543-448.

12. Druin, A. (1999). Cooperative Inquiry: Developing New Technologies for Children With Children. *In Proceedings of Human Factors in Computing Systems (CHI 99)* ACM Press, pp. 223-230.

13. Druin, A., Stewart, J., Proft, D., Bederson, B. B., & Hollan, J. D. (1997). KidPad: A Design Collaboration Between Children, Technologists, and Educators. *In Proceedings of Human Factors in Computing Systems (CHI 97)* ACM Press, pp. 463-470.

14. Myers, B. A., Stiel, H., and Gargiulo, R. (1998). Collaboration Using Multiple PDAs Connected to a PC. *Proceedings of Computer Supported Cooperative Work (CSCW 98)*, ACM Press, page 285.

15. Rekimoto, J. (1997). Pick-and-drop: a direct manipulation technique for multiple computer environments. *Proceedings of the 10th annual ACM symposium on User interface software and technology (UIST 97)*, ACM Press, pp. 31-39.

16. Rekimoto, J. (1998). A Multiple Device Approach for Supporting Whiteboard-Based Interactions. *Proceedings of Computer Supported Cooperative Work (CSCW 92)*, ACM Press, pp. 179-186.

17. Rekimoto, J., and Saitoh, M. (1999). Augmented surfaces: a spatially continuous work space for hybrid computing environments. *Proceedings of the CHI 99 conference on Human factors in computing systems: the CHI is the limit*, ACM Press, pp. 378 – 385.

18. Rubin, D. C. (1995). *Memory in Oral Traditions*. New York: Oxford University Press.

19. Singer, J., Behrend, S., Roschelle, J. (1988). Children's Collaborative Use of a Computer Microworld. *Proceedings of the conference on Computer-supported cooperative work, 1988,* ACM Press pp. 271-281.

20. Stanton, D., Bayon, V., Neale, H., Ghali, A., Benford, S., Cobb, S., Ingram, R., Wilson, J., Pridmore, T., & O'Malley, C. (2001). Classroom Collaboration in the Design of Tangible Interfaces for Storyteling. *In Proceedings of Human Factors in Computing Systems (CHI 2001)* ACM Press, pp. 482-489.

21. Stewart, J., Bederson, B. B., & Druin, A. (1999). Single Display Groupware: A Model for Co-Present Collaboration. *In Proc. Human Factors in Computing Systems (CHI 99)* ACM Press, pp. 286-293.

22. Stewart, J. (1999) Single Display Groupware. Doctoral Dissertation, University of New Mexico, Alburquerque, NM.

23. Strommen, E. (1994). Children’s use of mouse-based interfaces to control virtual travel. *In Proceedings of Human Factors in Computing Systems (CHI 94),* ACM Press, pp. 405-410.

24. Wood, D. & O’Malley, C. (1996). Collaborative learning between peers: An overview. *Educational Psychology in Practice*, 11(4), 4-9.