The Architecture and Implementation of CPN2000, A Post-WIMP Graphical Application

Michel Beaudouin-Lafon[*] *and Henry Michael Lassen*
Department of Computer Science - University of Aarhus
IT-Parken - Aabogade 34
8200 Aarhus N - Denmark
E-mail: mbl@daimi.au.dk, hml@daimi.au.dk

ABSTRACT
We have developed an interface for editing and simulating Coloured Petri Nets based on toolglasses, marking menus and bi-manual interaction, in order to understand how novel interaction techniques could be supported by a new generation of user interface toolkits. The architecture of CPN2000 is based on three components: the Document Structure stores all the persistent data in the system; the Display Structure represents the contents of the screen and implements rendering and hit detection algorithms; and the Input Structure uses "instruments" to manage interaction. The rendering engine is based on OpenGL and a number of techniques have been developed to take advantage of 3D accelerated graphics for a 2D application. Performance data show that high frame rates have been achieved with off-the-shelf hardware even with a non-optimized redisplay. This work paves the way towards a post-WIMP UI toolkit.

KEYWORDS: User interface toolkit, Advanced interaction techniques, Post-WIMP interfaces, Two-handed input, Instrumental interaction, OpenGL, Coloured Petri nets.

INTRODUCTION

Interaction techniques for desktop workstations have changed little since the creation of the Xerox Star in the early eighties. The vast majority of today's interfaces are still based on a single mouse and keyboard to manipulate windows, icons, menus, dialog boxes, and to drag and drop objects on the screen. While these WIMP interfaces are now ubiquitous, they are also reaching their limits: as new applications become more powerful, the corresponding interfaces become more complex. Since implementing new interaction techniques is notably difficult, interface designers and developers prefer to use the limited vocabulary offered by traditional widget sets.

A number of novel interaction techniques have been developed over the past decade, but few have found their way into commercial products. Toolglasses [9] and Marking menus [16] for example have been shown to be significantly more efficient than traditional palettes and menus, yet they are not available in commercial toolkits.

In order to foster the diffusion of post-WIMP interaction techniques into real applications, two problems must be addressed. The first is to study how these interaction techniques can be combined with each other and with more traditional techniques. This requires defining new interaction models, such as Instrumental Interaction [2] that encompass a wider range of interaction techniques. The second problem is to provide software support in the form of a toolkit or framework that allows application developers to incorporate post-WIMP interaction techniques as flexibly as they do with today's widget toolkits. Toolkits include an abstract interaction model and a set of modular interaction techniques. Two strategies for developing toolkits involve different trade-offs. Building the toolkit directly from the interaction model, with several demonstration applications, provides an early test of the ideas but raises questions about scalability and use within real applications. The alternative is to begin with a large-scale application based on the interaction model so as to identify the specific elements of the toolkit. This strategy permits testing the applicability of the model first, but risks compromising the generality of the toolkit when it is built.

The CPN2000 project

We chose the latter strategy and implemented the concept of Instrumental Interaction into the redesign of a complex application: CPN2000. The project is a complete redesign of Design/CPN, a graphical editor and simulator of Coloured Petri Nets (CPNs). Design/CPN has a standard WIMP interface, based on direct manipulation, menus and dialog boxes. It is in use by over 600 organizations around the world, in both academia and industry. Production CPNs can have over a thousand places, transitions and arcs, structured into a hundred modules or more.

The CPN2000 project started in March 1999. We used a highly participatory design process, involving the users throughout the design process [21]. Version 1 of CPN2000 was released in April 2000 and is in use by a small group of CPN designers for production work. We are conducting a longitudinal study to inform the iterative design process.

in Proc. ACM Symposium on User Interface Software and Technology (UIST 2000), San Diego, CA, November 2000, ACM Press, in press.

[*] Author's current address: LRI, Bât 490, Université Paris-Sud, 91405 Orsay Cedex, France. Email: mbl@lri.fr

The CPN2000 interface uses a combination of traditional, recent and new interaction techniques, e.g. tool palettes, toolglasses, and magnetic guidelines. Integrating these interaction techniques together in a consistent way in a single tool proved quite challenging. We wanted to design a system that would strike a better balance between power and simplicity than current WIMP interfaces. This led us to define three design principles: reification, polymorphism and reuse [1]. Reification states that any entity in the interface should be accessible as a first-class object. Polymorphism states that commands should apply to as many different types as possible. Reuse states that any output generated by the system and any input to the system should be reusable later, e.g. in the form of macros.

The resulting interface has no menu bars, no pull-down menus, no scrollbars, no dialog boxes and no notion of selection. Instead, it uses a unique combination of floating palettes, toolglasses and hierarchical marking menus, a novel windowing model based on pages and binders, and several new interaction techniques such as magnetic guidelines to align objects and bi-manual interaction to manipulate objects. This interface supports all the functions of the previous Design/CPN application and more, yet we have empirical evidence [21, 1] that it is both simpler to use and more powerful. Implementing this interface required that we started from scratch: most toolkits do not support our graphical model (e.g. transparency and non-rectangular windows) nor our input model (bi-manual input). We decided to use OpenGL for output, and to design and implement our own input management, based on the Instrumental Interaction model [2]. The next section gives an overview of Coloured Petri Nets and the CPN2000 interface (see [1] for more details).

THE CPN2000 INTERFACE

Coloured Petri Nets

A Petri net is a bipartite graph with nodes called *places* (depicted as circles or ellipses) and *transitions* (depicted as rectangles). Edges of the graph are called *arcs* and can only connect places to transitions and transitions to places. Places hold tokens that represent the current state of the system. Simulating the net involves "firing" transitions to move tokens from place to place.

Coloured Petri nets [14] are an extension of Petri Nets for modeling complex systems. Tokens belong to *color sets* similar to data types in programming languages. Arcs are labeled with pattern-matching expressions that describe which tokens are used when a transition is fired. Typically, colors allow a conventional Petri net to be "folded" onto itself, making models much smaller. In addition CPNs are hierarchical. A transition can be described by a subnet, equivalent to macro-substitution in a textual language.

Graphically, a CPN looks like a traditional Petri net, with additional text inscriptions on the places, transitions and arcs. A CPN may consist of several diagrams. Graphical annotations describe how the diagrams compose the net.

Overall interface

The CPN2000 interface requires a traditional mouse and keyboard, plus a trackball (or other locator) for the non-dominant hand. For simplicity, we assume a right-handed user, but the mouse and trackball can be swapped for left-handed users. A large window, called the *workspace*, holds an index to the left, a set of floating palettes, which can be turned into toolglasses, and a set of window-like objects called *binders* that contain *pages* (figure 1). The pages in a binder are accessible by tabs similar to those found in tabbed dialogs. A page can be moved within or between folders by dragging its tab, or moved to the workspace, which creates a new binder holding it. The index contains an entry for each tool palette. When loading a CPN model, a new entry is created in the index, with a sub-entry for each diagram in the model. Dragging entries of the index onto the workspace creates one of the objects above (floating palette, toolglass or page within a binder).

Interaction techniques

Floating palettes are similar to those found in traditional interfaces: clicking a tool activates it. The tool is then held in the right hand and applied by clicking or dragging. Unlike traditional interfaces, there is no notion of selection and therefore no selection tool. This avoids the problem of unwanted switching to the selection tool found in traditional graphical editors. To deactivate the tool in hand, one clicks on the tool again in the tool palette.

Direct manipulation (i.e. clicking or dragging objects by clicking them directly) is used for frequent operations such as moving objects, panning the content of a view and editing text. When a tool is held in the right hand, direct manipulation actions are still available via a long click, i.e. pressing the mouse button, waiting for a short delay (300ms) until the cursor changes, and then either dragging or releasing the mouse button. Because of the visual feedback, this multiplexing of tools in the right hand is easily understood by users.

Bi-manual manipulation is a variant of direct manipulation that involves using both hands for a single task. It is used to resize objects (binders, places, transitions, etc.) and to zoom the content of a page. The interaction is similar to holding an object with two hands and stretching or shrinking it. Bi-manual interaction could also be used to control the orientation and position of an object as in T3 [18]. This might be used in the future to control the orientation of our magnetic guidelines.

Toolglasses [9] are positioned with the left hand and operated by a click-through with the right hand. Click-through tools take precedence over the tool that may be held in the right hand; however the long-click manipulations are still available. When the click-through tool requires a drag action to specify an object size, the toolglass disappears while the mouse is dragged. The toolglass is turned into a palette, and vice versa, by clicking the right button on the trackball (left hand). We call these actions picking up and dropping the toolglass.

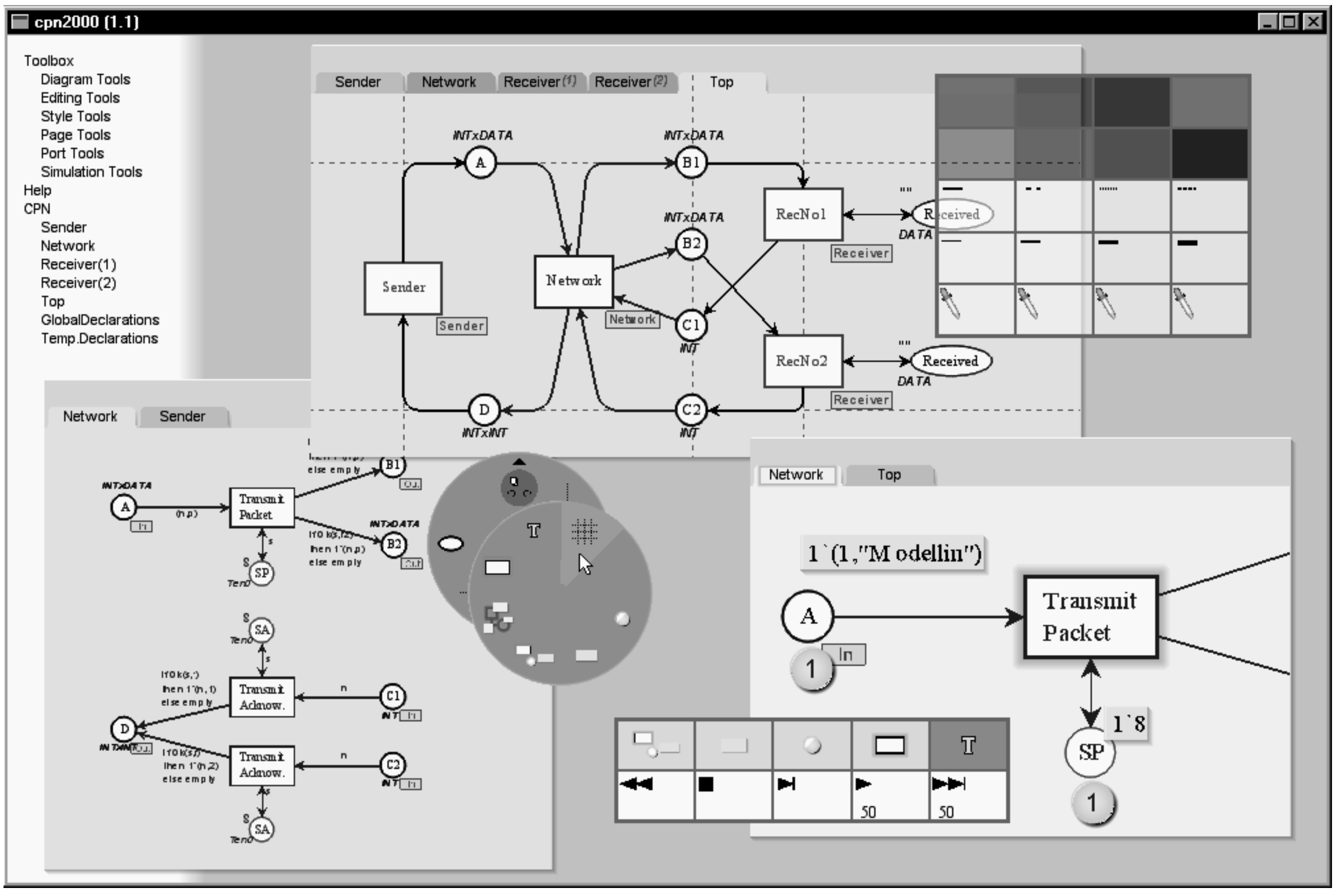

Figure 1: The CPN2000 interface, showing the index (top-left), three binders with multiple pages, a tool palette (bottom), a toolglass (top-right) and a hierarchical marking menu (center). The top page has magnetic guidelines. The bottom binders show two views of the same diagram at different sales, with simulation information in the right one.

Marking menus [16] are available throughout the interface by clicking the right mouse button. The commands in these contextual marking menus are also available in palettes/toolglasses. Our marking menus have at most eight entries per menu and at most one level of sub-menus.

Magnetic guidelines [1] are used to align objects. They combine some aspects of snap-dragging [12] and graphical tabs [7]. Objects snap to a guideline when moved close to it, and can be detached by dragging them away. Moving the guideline moves the objects attached to it. The current system supports horizontal and vertical guidelines. Objects can be attached to both simultaneously.

Keyboard input is used only to edit text. Some navigation commands are available at the keyboard to make it easier to edit several inscriptions in a row without having to move the hands to the mouse and trackball. Keyboard modifiers and shortcuts are not necessary since most of the interaction is carried out with the two hands on the locator devices.

An important characteristic of the interface is that it supports multiple working styles. Tool palettes are efficient when a single tool needs to be applied to multiple objects; marking menus are more efficient when multiple commands are applied to the same object in succession; toolglasses support a mix of these and are quite efficient for creating a structure with different types of objects (places, arcs, transitions), and for editing the graphical attributes (color, thickness) of a set of related objects, e.g. around a cycle.

SOFTWARE ARCHITECTURE

The system architecture is depicted in figure 2. The Document Structure holds a set of Abstract Documents that represent all persistent data in the system. The Abstract Petri Net is the part that represents the CPN diagrams and interfaces with the CPN simulator. The simulator runs as a separate process and is not covered in this description. The Display Structure is used for rendering the document structure and for hit detection. The Input Structure manages the interaction instruments that edit the documents.

The Document Structure

Abstract documents represent the persistent data of the system, including the diagrams being loaded, the tools and tool palettes, and the configuration of the workspace. This structure must be both extensible and efficient. The file format for storing the data in abstract documents is XML.

An abstract document is a collection of typed nodes. Since a common operation is to enumerate all the nodes of a type *T*

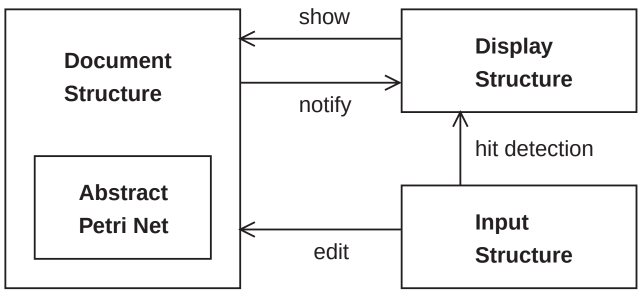

Figure 2: Main components of the architecture

or any of its subtypes, an abstract document dynamically builds a representation of the hierarchy of the types of the nodes it contains. Each node is stored in a list attached to its type node. Objects of type *T* are enumerated by walking down the tree rooted at the node representing type *T* and enumerating all the nodes attached to it and its sub-nodes.

Nodes in abstract documents can be organized in a graph by creating typed associations among them. The graph can then be traversed in a number of ways. Any other structure could be superimposed on a document. For example, a spatial index such as in Pad++ [4] could be used to optimize access by the display structure.

Abstract documents generate notifications when nodes are added, removed or changed. The observers of these notifications are the items of the display structure described below as well as other nodes in the document structure. For example, an arc needs to know when its place or transition is moved or destroyed.

Abstract documents are used to represent interface elements such as pages, binders, tool palettes and marking menus. This means that instruments are first class objects that can be stored in XML files and edited like any other object. This also supports the customizability of the interface.

The Display Structure

The role of the display structure is to represent what is on the screen. It is used both by the rendering algorithm to update the screen and by the input structure for hit detection. The display structure is a special scene graph, similar to those found in 3D graphics toolkits such as Inventor [26]. Typically, a 3D scene graph is a tree or DAG (direct acyclic graph) of nodes. Some nodes represent state changes, e.g., set the current color, while others represent geometry information, e.g., draw a cube. Scene graphs tend to be large and not optimized for rendering. Their structure reflects the needs of the application rather than those of the rendering algorithm.

Our display structure (figure 3) is slightly more abstract than traditional scene graphs because we separate the document structure from the display structure. Therefore the document structure can be organized to suit the needs of the application while the display structure can be optimized for rendering and hit detection.

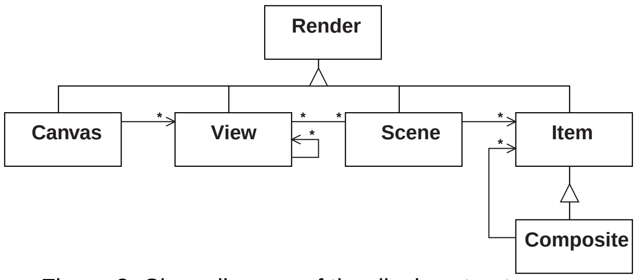

Figure 3: Class diagram of the display structure

The root of the structure is a Canvas, representing the whole screen. The Canvas contains an ordered set of Views, stacked from back to front. Views correspond roughly to windows in traditional window systems. Three special views are always present: a background view (behind any other view), a foreground view and an overlay view (both in front of any other view). The overlay view takes advantage of hardware overlay planes, when available, otherwise it is the same as the foreground view. It contains mouse and trackball cursors and other lexical feedback.

Each view may contain other views, for grouping. Unlike hierarchical window systems, a sub-view is not enclosed in (and clipped to) its parent view, unless the parent view has clipping enabled. Views can be semi-transparent, i.e. the views underneath them may show through, and they can have any shape. This is how we support toolglasses.

The contents of a View is one or more Scenes, which are stacked like overlapping layers as in, e.g., Xtv [3] or the multi-layer model [10]. The Scene is not necessarily enclosed in (and clipped to) its view, unless the view has clipping enabled. We use different scenes to represent a single document in order to control which parts of the document structure is visible in a page, such as:

- *main layer*: places, transitions and arcs, no textual inscriptions except the names of places and transitions;
- *text layer*: all text except names of places and transitions;
- *guidelines*: magnetic guidelines for aligning objects;
- *simulation*: tokens and messages related to the simulation
- *annotations*: error messages, help tips, etc.

A Scene can be shared among multiple Views. For example, two pages may display the same diagram by sharing the main layer and the text layer. One page may show the guidelines layer while the other displays the simulation layer, and each page may use a different scale.

Finally, the contents of a Scene is a set of Items. The Items can be organized in a list, a tree or a DAG. Items are meant to be lightweight objects: they typically contain a reference to the node of the document structure that they represent, and some information used solely by the redisplay and hit detection algorithms. Since an observer design pattern is used to notify the display structure of changes in the document structure, multiple items can reference the same document node. This supports multiple graphical representations (as opposed to multiple views) of the same document. For example, a CPN diagram can be shown as a graph in one page and as a list of objects in another page by using two scenes and two sets of items referencing the same nodes in the abstract document, one for each representation.

The Input Structure

In most modern toolkits, input management is based on an event-driven model where input events are dispatched to widgets based on their location or some global state. Widgets respond to events by invoking callbacks. Callbacks are known to complicate programming [23] because they create dependencies in the code that are not reflected in its structure.

Our input structure is based on the Instrumental Interaction model [2]. An *interaction instrument* is the association of a physical part (the input device) and a logical part (the representation on the screen). An instrument operates on a node of the document structure called the *target*. When receiving input events, an instrument provides feedback by updating its on-screen representation and sends commands to its target. The target responds to these commands by updating its own state.

In the widget model, objects that want to respond to an interaction with a widget need to set callbacks on that widget. In our model, the instruments select a target and send commands to it. This makes it easy to create new instruments that operate on existing objects without changing them. For example, it took less than two hours to create a help instrument that works with any object in the interface (including other instruments) and that is available from contextual menus as well as in a tool palette.

Some instruments are *generic*. A generic instrument contains "concrete" instruments that represent implementations of the generic instrument for different target types. When a generic instrument is activated, it looks at the target object type and activates the concrete instrument that corresponds to that type. For example the move instrument is a generic instrument, with many concrete instruments for moving places, transitions, bend points of an arc, inscriptions, magnetic guidelines, pages, binders, entries in the index, etc.

The representation of an instrument is itself an abstract (but usually very simple) document. As explained before, this makes it easy to edit instruments and load them from a file. It also means that instruments can themselves be the targets of other instruments. This is used to support interactive customization of the interface. For example, a move instrument can be used to drag a tool from one palette to another. Finally, instruments, like other documents, can have multiple graphical representations. This allows the same instrument to be used in a palette and a menu.

Several instruments may be ready for use at any one time. In our design, the user has instant access to up to five instruments at a time:

- the tool selected in a palette and held in the right hand;
- the move/pan tool accessible with a long click with the right hand, or with a short click when no tool is selected;

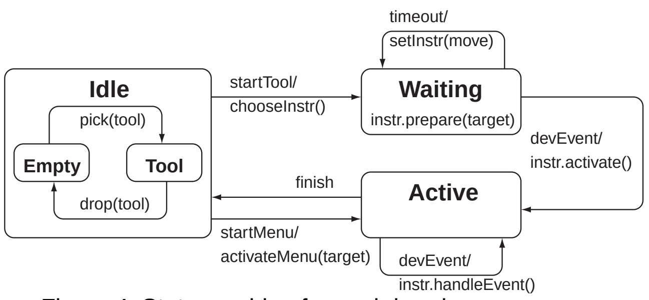

Figure 4: State machine for each hand

- the marking menu accessible with the right mouse button
- the toolglass that can be moved with the left hand and clicked-through with the right hand;
- the resize/zoom tool accessible by bi-manual interaction (left button press on trackball then click and drag mouse).

The input manager uses two state machines to manage the activation of instruments, one for each hand (figure 4). The transitions of the state machine are abstract events generated by the input devices or the instrument. The *Idle* state has two sub-states, *Empty* and *Tool*, describing whether a tool has been selected (right hand) or a toolglass has been picked up (left hand). Clicking the right mouse button generates a *startMenu* event and goes into the *Active* state. Clicking the left button on the mouse or trackball generates a *startTool* event. According to the context, an instrument and target are selected (*chooseInstr*), the state machine enters the *Waiting* state and the instrument is informed that it may be activated (*prepare*). If a *timeout* occurs, a long click has been detected and the move instrument replaces the previously-selected instrument. Other events cause the machine to enter the *Active* state, where the instrument receives all events. It interacts with its target by sending it commands. Commands are first-class objects that can be stored in a history, done, undone, and redone. Executing a command normally results in some edit of the document structure, which will be reflected on the screen at the next redisplay. At some point the instrument generates a *finish* event and the state machine is reset.

Since there are two state machines, one for each hand, it is possible to execute parallel actions, e.g., moving an object with the right hand while panning the contents of a page with the left hand. For interactions that involve both hands, such as resizing and zooming, one state machine sets the state of the other to *Active* (if it is *Idle*) and passes it its instrument and target. The two state machines therefore send their respective events to the same instrument. A future version of the tool will support single-display groupware (see, e.g., MMM [8] or KidPad [6]) by assigning one state machine per user hand.

USING OPENGL FOR RENDERING

We decided to use OpenGL [27] as our rendering engine for several reasons. First, the performance of 3D accelerated graphics is increasing faster than Moore's law, thanks to applications such as video games driving the market, and more and more of the rendering costs are off-loaded to the graphics hardware, saving CPU cycles. Second, OpenGL is a standard API, implemented by many vendors on the three main platforms in use today: Windows, MacOS and Unix. Third, OpenGL has a sophisticated graphics model that supports transparency, lighting and texturing, opening the door to advanced visual effects.

OpenGL can be seen as a state machine that processes two types of requests through a rendering pipeline: geometry requests are transformed into pixel values in a frame buffer, and state requests affect the state of the rendering pipeline. The rendering pipeline may be partially or completely implemented in hardware, dramatically improving its performance. Using OpenGL for rendering complex 2D scenes has some important implications. First, we cannot use the host window system to implement the windows in our interface because toolglasses and mark-based interaction require drawing outside the document windows. Second, the OpenGL API is designed for applications that redisplay the whole frame buffer whenever a change is made. This is because, in 3D, even a small change in the position of an object (e.g. a light or the camera) changes most of the display. In addition, graphics hardware uses double-buffering to avoid flicker: the application displays the frame in the back-buffer while the front-buffer is being computed, then swaps the front and back buffers. No hypothesis can be made in general on the content of the back buffer after the swap, so the application must redraw it from scratch.

These characteristics led us to a radical decision: we run the application in one large OpenGL window and implement our own windowing system inside that host window, called the screen; the rendering algorithm redisplays the whole screen at each frame. Unlike all commercial 2D applications and toolkits and most of the research ones, we do not implement an incremental redisplay algorithm, and we do not use the host window system. This dramatically simplifies the implementation of the rendering structure. As shown in the performance data below, this approach proved to work, and increase in graphics hardware performance will make it more and more valid. It also means that interaction is always smooth, whether the user just types in some text or moves pages around while the simulation is running. The rest of this section explains the most important aspects of the rendering and hit detection algorithms.

Windowing and Clipping

Traditional window systems clip objects to their parent window so they do not extend outside of it. OpenGL does not provide windows so the rendering algorithm implements its own clipping. We review three clipping methods and present the one we have created.

Analytical clipping computes the visible part of each object. This is often impractical, and graphics hardware is of little help, if any, to make it efficient. A crude form of analytical clipping consists in eliminating the objects that are not visible at all, e.g. with a bounding box test, and use another clipping technique for the objects that might be visible. In our case, this could be used to insert in the display structure only those nodes that may be visible. This is not currently implemented in our system.

Using the depth buffer: The depth buffer is used for hidden-surface removal in 3D applications and is implemented in hardware on all recent graphics cards. It records the depth value (z coordinate) of each pixel in the frame buffer so that it can decide whether a new pixel is visible or not. Clipping can be achieved with the depth buffer by drawing a shape consisting of the whole screen with a hole for the window at $z=z0$ and all the objects inside the window at $z < z0$. This is inefficient because drawing large areas is expensive and it makes it hard to use the depth buffer for anything else.

Scissoring is a stage in the OpenGL rendering pipeline where the graphics primitives are clipped against a rectangle parallel to the axes. This technique is very efficient, but only applicable when the clipping shape is a rectangle.

Using the stencil buffer: The stencil buffer is less common than the depth buffer on current hardware, but is available on all middle and high-end graphics card. The stencil buffer holds a value for each pixel of the frame buffer. When stenciling is enabled, each pixel is tested against the stencil buffer just before going through the depth test. For each pixel *p(x,y)*, the stencil test compares the value *s(x,y)* in the stencil buffer with a reference value *r*. Depending on the result of the comparison *s(x,y) TEST r*, the pixel passes the stencil buffer or is not drawn. In addition, depending on the result of the test, the content of the stencil buffer may be changed according to an operation *OP*: It can be incremented, decremented, or set to the reference value *r*.

To implement clipping, we assign a unique ID to each clipping shape and we draw them in the stencil buffer with their respective IDs. We call the resulting stencil buffer a *clip buffer*. The algorithm is as follows:

```
clear the stencil buffer with value 0
id <- max
enable stencil test with TEST="<" and OP="set"
for each clipping shape from front to back
    set stencil reference to r=id
    draw the clipping shape
    id <- id - 1
```

When the shape with ID *id* is drawn, the only pixels that pass the stencil test are those for which *s(x,y) < id*. Since previous shapes have been drawn with a greater ID, only the pixels with value 0, i.e. the background, will pass the test. In other words, only the visible part of the shape will pass the stencil test and will be drawn. And since only the pixels that pass the test update the stencil buffer, the new ID will be written in the stencil buffer only where the shape is visible. This seems convoluted, but the only way to change the content of the stencil buffer is by drawing pixels that pass the stencil test.

At the end of the above algorithm, the visible part of each clipping shape is "painted" with that shape's ID in the clip buffer. To clip an object to a shape with ID *id*, stenciling must be enabled with a test *s(x,y)=id* before the object is drawn. In order to minimize the number of changes to the stencil reference value, all the objects clipped by the same shape should be displayed together.

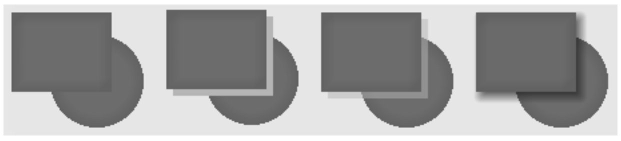
Figure 5: Using a gradient fill to contrast objects (left); Hard shadow: opaque (middle left) and transparent (middle right) ; Transparent soft shadow (right)

Note that the above algorithm also draws the clipping shapes in the frame buffer, which would have to be done with the other techniques as well, so its cost is minimal. Also, recreating the clip buffer is unnecessary as long as the clipping shapes do not change. Finally, the above algorithm also works when the shapes are drawn in any order, provided that they have been assigned decreasing IDs from front to back. Since a stencil buffer is typically 4 to 8 bits deep, the maximum number of clipping shapes that can be simultaneously dealt with is between 15 and 255. It is possible to support an arbitrary number of clipping shapes by drawing them and their contents by batches and clearing the stencil buffer between each batch.

Two extensions to the above clipping algorithm are necessary to support semi-transparent clipping shapes and hierarchical clipping shapes. Transparency will be discussed in the next subsection. To implement hierarchical clipping shapes, we assign the IDs so that the ID of a shape is greater than the ID of each of its sub-shapes and each of the shapes behind it. This is easily accomplished by a preorder traversal of the tree of shapes. Drawing the clip buffer is accomplished by a post-order traversal so that each clipping shape is drawn before it's sub-shapes, from front to back.

Displaying text and graphics

OpenGL is designed for 3D graphics and is specially optimized for displaying large numbers of small convex polygons (triangles or convex quads). This can be quite challenging for displaying the type of graphics found in 2D applications. For example, support for drawing lines is minimal, and quite flaky on some graphics cards, and there is no support for drawing text.

We have developed a library of simple 2D shapes, including arbitrary polygons and polylines with straight and Bezier segments, arrows, and ellipses. Each shape has a thickness and a different color and transparency can be specified for the center and the border of the shape, resulting in a gradient fill (figure 5). Each shape can be drawn in fill, border or shadow style. Fill style draws the regular shape, border style draws its outline (the thickness of the outline can be specified), and shadow style draws a soft shadow of the shape (the offset and transparency of the shadow can be specified).

Soft shadows are particularly effective to convey the impression of overlapping objects. We use them for the borders of the pages and binders and as feedback when an object is moved. Soft shadows are much more convincing than hard shadows, but harder to compute. Computing exact soft shadows requires a convolution operation [27] between the shape of the light and the shape generating the shadow.

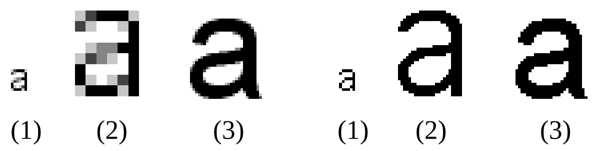

Figure 6: Scaling (2) vs. re-generating a font (3) when zooming textured text (left) and outline text (right)

This is expensive because very few graphics cards implement convolution in hardware. We use an analytical approximation of soft shadows that corresponds to a circular light source (figure 5) by computing a shape that is a fixed distance from the reference path.

The three approaches to displaying text include: bitmaps, outlines and textures. We use outlines and textures, depending on the point size of the font: using textures gives better results but uses up too much texture memory for larger point sizes, and the difference in the quality of rendering between textures and outlines is hardly noticeable for larger point sizes. We use the Freetype engine (http://www.freetype.org) to generate both the outlines and the anti-aliased bitmaps of any TrueType font. Outlines are turned into sets of triangles by a tessellation algorithm, while bitmaps are turned into textures. In both cases, each character is stored in an OpenGL display list, i.e. a sequence of OpenGL requests accessible through a single call.

Text rendering dramatically affects the overall performance of the rendering algorithm. On most graphics hardware, enabling and disabling texturing is very expensive, and swapping textures in and out is even more expensive. We use caching and grouping to optimize the rendering of text: several textures are kept in memory to minimize swapping textures in and out; text items that use the same font are drawn together when possible. When using textures, the text looks better if the font has been rendered by the font engine at the apparent point size, i.e. the actual point size multiplied by the zooming factor. This is because TrueType fonts contain hints to improve the readability of the characters at small point sizes. Since generating a new font is fairly expensive (1 to 2 seconds), it is not done during zooming operations. Instead, text is scaled geometrically. At the end of the interaction, the fonts are re-generated and the text is redisplayed (figure 6). We plan to implement a font cache on disk to speed up this operation.

The drawing model for 2D graphics is often described as 2D1/2: objects have a 2D geometry but have a rank order to determine which one is in front when they overlap. Rendering this type of scene can be done with the traditional painter algorithm by displaying the objects from back to front. With a 3D graphics model, we can take advantage of the depth-buffer to display objects in any order and yet get the correct result. All we need to do is assign a z-value to each object consistent with its rank. If the scene contains semi-transparent objects, e.g., the soft shadows described above, the drawing order is more constrained: each semi-transparent object must be drawn after any object it overlaps. In addition, if semi-transparent objects overlap, it is better to draw them from back to front, although the difference is often hardly noticeable. Finally, the depth-buffer must not be modified when transparent objects are drawn, but the depth-test must be enabled so that the semi-transparent objects that are behind opaque objects are rendered correctly. The rendering algorithm is:

```
draw all opaque objects in any order
if there is at least one semi-transparent object
    disable depth-buffer writes
    draw all semi-transparent objects,
        preferably from back to front
    enable depth-buffer writes
```

Optimizing redisplay

As noted above, the depth-buffer lets us draw the objects in a scene in any order, so we can pick an order that optimizes performance. The problem is that the best order may depend on the graphics hardware. On current graphics hardware, some state change requests are very expensive. For example, on our reference platform, enabling and disabling texture mapping for each character, as opposed to each string, degrades performance by a factor of 2. Therefore, it is preferable to draw objects that use textures together, and it is preferable to draw objects that use the same texture (i.e. the same font at the same apparent point size) together. So far we have only implemented these optimizations in the display structure rendering on a per-view basis.

Another way to improve performance is to use *display lists*. A display list stores an arbitrary sequence of OpenGL requests and can be called with a single request. A display list can call another display list, so the structure of display lists is a tree or DAG. In the current implementation, we do not take advantage of display lists except for drawing text. The performance evaluation below shows that up to 50% of the redisplay time is spent creating a large display list that represents the whole screen. By breaking this list into a hierarchy of smaller ones, only the display lists that are affected by a change in the document structure need to be rebuilt. Since most interactions change only a few nodes in the document structure and therefore impact very few items in the display structure, this strategy should reduce the time to rebuild the display lists to almost zero, therefore doubling the performance.

The last optimization technique we have implemented is the use of the overlay buffer for lexical feedback. The overlay buffer is a separate frame buffer that is superimposed over the main frame buffer by the video system. Overlay buffers are not always available and when they are, they have a limited number of planes (from 2 to 8). When the overlay buffer is available, we use it for displaying the mouse and trackball cursors and the feedback of the ink when using gestures to select an entry in a marking menu. This saves redisplays of the main frame buffer. Otherwise, the lexical feedback is displayed in the main frame buffer.

Hit detection

The display structure is used both for rendering and for hit detection, i.e. to determine which object is at a given position on the screen (typically the cursor position). Application frameworks and user interface toolkits traditionally implement hit detection by walking through data structures to find out, analytically, which object is under the given position. Indeed, we could have implemented the same technique since we have all the relevant information in the display an document structures. However we decided to use an alternative approach based on OpenGL's select mode. When the rendering pipeline is in select mode, it does not draw into the frame buffer. Instead, it constructs and returns a *select buffer* that contains the information about all the object that have been drawn and that intersect the *viewing rectangle*. By setting the viewing rectangle to a tiny window around the cursor position, we can find which objects are at or near the cursor. To optimize this technique, we use a simplified drawing method for the items when in select mode. This method does not set the attributes such as color and texture that do not affect the result of picking. This significantly speeds-up the redisplay.

One problem with using select mode is that hit detection occurs *before* the stencil and depth tests in the graphics pipeline. As a result, the invisible part of a clipped object may be reported in the select buffer, and all objects at a given position, not just the topmost one, are reported. To address the first problem, we use two passes: the first pass renders the clipping shapes only so we know which shape (if any) was hit; the second pass renders only the objects contained in that shape or, if no shape was hit, the objects that are outside any clipping shape. This actually reduces the number of rendered objects and speeds up hit detection. The second problem is easy to solve since the z-coordinate of the objects is stored by OpenGL in the select buffer, so we just need to consider the object with the smallest z. Knowing all the objects at a given position in the hit buffer is actually useful when clicking through a toolglass since it allows to identify the tool *and* the clicked-through object.

PERFORMANCE EVALUATION

The data reported in this section corresponds to version 1 of CPN2000, which is fully functional and has been in use by a small group of members of the CPN group for real work for several months. Our reference platform is a HP/Kayak XW with a 500 MHz Pentium II and a Visualize FX6 graphics card running Windows NT4. We also use a similar PC with a Diamond FireGL 1 graphics card and a SGI Visual PC 320 for testing. CPN2000 also runs on Unix (SGI O2 and Octane) and Macintosh.

The system is implemented in Beta [19], a high-level, compiled, strongly-typed object-oriented language based on a single construct, the *pattern*, that unifies classes, objects and methods. Beta is taught to all the students at University of Aarhus, therefore all the programmers in the project were fluent in Beta. However, most of them had no prior knowledge of OpenGL, computer graphics, or implementing highly interactive applications. The implementation consists of 40000 lines of Beta code.

We have conducted a preliminary evaluation of the memory footprint and display rates of the system. However, we must stress that virtually no optimization has been carried out on this version. The memory footprint is typically between 10 and 12Mb when a medium-size CPN model is loaded, compared to 7.5Mb for the former Design/CPN tool. A break-down of memory usage shows that the Document, Display and Input structures use only 3Mb of the total, plus 1Mb for the texture cache. 2Mb are used by the Beta runtime for garbage collection, and the remaining 4Mb are used by OpenGL and the rest of the Beta run-time. While loading fonts, the Freetype engine may use up to an additional 10Mb. The above-mentioned font cache could save most of that overhead.

The table below summarizes the number of graphical objects (polygons and characters) and the frame rates for typical situations on the reference platform. The size of the display window is 1280x1024, and all measurements are made with a semi-transparent toolglass on the screen.

Content of Display	#objects	fps
Empty workspace	91	50
One binder with five pages	559	22
Two binders with five pages each	941	15
Five binders with one page each	2714	12
Five binders with text layer disabled	702	16

A frame rate of 15 fps (frames per second) or more is smooth enough for our interaction techniques, including moving pages and toolglasses, and panning or zooming their content. A close analysis of the profiling information shows that 50% of redisplay time is currently spent re-building the global display list that holds everything on the screen. (See the solution to this problem presented above.) Also, profiling data shows that displaying text is time-consuming, as shown by the last two lines in the table above. But our strategy of full-screen redisplay of a large OpenGL window is clearly successful and the increase in performance of graphics cards will only make things better: the FireGL1 came out less than one year after the Visualize FX6, costs a fraction of its price and is 30% faster!

SUMMARY AND DISCUSSION

The key property of the CPN2000 architecture is its flexibility. The Document Structure is a "soup" of nodes on top of which the application can create its own structure. It unifies the representation of application data and user interface elements, allowing the interaction instruments and the rendering algorithm to work with both. The Display Structure supports a rich graphics model, including multiple views, multiple layers, transparency, arbitrary clipping shapes, high-quality text and soft shadows. The rendering algorithm already exhibits high frame rates but still can be improved. Finally, the Input Structure implements a complex combination of interaction techniques and can easily support multiple simultaneous users (single-display groupware). Moreover, interaction instruments can be created independently of the objects they operate on, in contrast with widget-based toolkits.

Even though CPN2000 is *not* a toolkit, it can be compared with the current state of the art in user interface toolkits. Jazz [5] is based on a scene-graph technology close to ours although, like Inventor [26], it does not clearly separate the document and display structures. Unlike InterViews [20], Pad++ [4] and Amulet [22], we do not implement an incremental redisplay but use hardware accelerated graphics to render full frames at each redisplay. Toolglasses and Magic Lenses have been implemented in different environments (Cedar [9], X [23], and SubArctic [13]) with different methods. We use transparency for Toolglasses; Magic Lenses are not supported yet but can be implemented within our display structure with a variant of the Model-In Model-Out technique [9].

Amulet [22] and SubArctic [13] rely heavily on one-way constraints to describe the dependencies in the system while we use only a simple observer pattern to propagate changes in the document structure. We will need a constraint solver for layout to support more general magnetic guidelines, but we believe that a specialized solver will be easier to implement and more efficient than a general one.

CPN2000 exhibits a unique combination of interaction techniques, an area with little previous work, to our knowledge (see [17] for an example). KidPad [6] supports multiple input devices with combinations of actions similar to our bi-manual techniques. Hinckley uses a single Petri net [11] to describe two-handed input, while we combine simpler state machines. Myers' interactor model [24] was designed for widget-based toolkits, and has not been used for bi-manual interaction or toolglasses. We discovered that combining interaction techniques was tricky and cannot be done in general by simply juxtaposing them. Therefore additional work is necessary to facilitate such combinations.

Our architecture model is quite different from MVC [15]. Even though the documents can be seen as models, the display structure as a set of views and the interaction instruments as controllers, there is no equivalent to the MVC triplets that constitute a traditional Smalltalk application. This is because the associations between these three structures change over time, while MVC triplets are mostly static. For example, the link between an instrument and its target node in the document structure changes each time the instrument is activated. This approach eliminates the need for callbacks by localizing the code that manages an interaction within the interaction instruments. This dramatically simplifies the development of the system.

CONCLUSION AND FUTURE WORK

We have described the architecture and implementation of CPN2000 and shown how it supports a combination of advanced interaction techniques in a post-WIMP interface. Our experience with the framework during the multiple iterations of the design process, makes us confident that it can be reused to create graphical interfaces based on the same or a different set of interaction techniques.

The next step will be to turn this framework into a separate toolkit. This toolkit must include a large collection of traditional and post-WIMP interaction techniques, additional services such as animation and layout, and a richer set of graphical objects and rendering effects. More challenging is the execution of several applications within the same OpenGL context, e.g., to mix CPN diagrams, text documents and spreadsheets in the workspace and support instruments that work across these document types. This probably requires a client-server architecture similar to that of X Windows but where the server would provide high-level interaction services, not just windowing and graphics. This would be a significant step towards a document-centered environment and an alternative to current desktops.

ACKNOWLEDGMENTS

CPN2000 is a team effort: Our thanks to Kurt Jensen, Søren Christensen, Wendy Mackay, Peter Andersen, Paul Janecek, Mads Jensen, Kasper Lund, Kjeld Mortensen, Stephanie Munck, Katrine Ravn and Anne Ratzer for their work on the design and implementation of the system and their enthusiasm for the project. We are also indebted to the members of the CPN group at the University of Aarhus for their participation in the design and evaluation activities. Project URL: http://www.daimi.au.dk/CPnets/CPN2000.

This work is supported by the University of Aarhus, the Danish Centre for IT Research (CIT), Hewlett-Packard and Microsoft Research.

REFERENCES

1. Beaudouin-Lafon, M. & Mackay, W. Reification, Polymorphism and Reuse: Three Principles for Designing Visual Interfaces . In *Proc. Conference on Advanced Visual Interfaces*, AVI 2000, Palermo, Italy, May 2000, ACM Press, 2000, p.102-109.
2. Beaudouin-Lafon, M. Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces. In *Proc. Human Factors in Computing Systems, CHI'2000*, ACM Press, 2000.
3. Beaudouin-Lafon, M., Berteaud, Y., Chatty, S. Creating direct manipulation applications with Xtv. *Proc. European X Window System Conference*, EX'90, London, Nov 1990.
4. Bederson, B. & Meyer, J. Implementing a Zooming Interface: Experience Building Pad++. *Software Practice and Experience*, 28(10):1101-1135, August 1998.
5. Bederson, B. B. & McAlister, B. Jazz: An Extensible 2D+Zooming Graphics Toolkit in Java. Tech Report HCIL-99-07, Computer Science Department, University of Maryland, College Park, MD.
6. Benford, S., et al. Designing Storytelling Technologies to Encourage Collaboration Between Young Children. In *Proc. ACM Human Factors in Computing Systems*, CHI 2000, ACM Press, 2000, p. 556-563.
7. Bier, E. & Stone, M. Snap-dragging. In *Proc. ACM SIGGRAPH*, ACM Press, 1986, 20(4):233-240.
8. Bier, E. & Freeman, S. MMM: A User Interface Architecture for Shared Editors on a Single Screen. In *Proc. ACM Symposium on User Interface Software and Technology*, UIST'91, ACM Press, p. 109-118.
9. Bier, E., Stone, M., Pier, K., Buxton, W., De Rose, T. Toolglass and Magic Lenses : the See-Through Interface. In *Proc. ACM SIGGRAPH*, ACM Press, 1993, p. 73-80.
10. Fekete, J-D. & Beaudouin-Lafon, M. Using the Multi-layer Model for Building Interactive Graphical Applications. In *Proc. ACM Symposium on User Interface Software and Technology*, UIST'96, ACM Press, p. 79-86.
11. Hinckley, K. , Czerwinski, M., Sinclair, M. Interaction and Modeling Techniques for Desktop Two-Handed Input. In *Proc. ACM Symposium on User Interface Software and Technology*, UIST'98, ACM Press, p. 49-58.
12. Hashinoto, O & Myers, B. Graphical Styles for Building User Interfaces by Demonstration. In *Proc. ACM Symposium on User Interface Software and Technology*, UIST'92, ACM Press, p. 117-124.
13. Hudson, S. & Smith, I. Ultra-Lightweight Constraints. In *Proc. ACM Symposium on User Interface Software and Technology*, UIST'96, ACM Press, p. 147-155.
14. Jensen, K. *Coloured Petri Nets: Basic Concepts (Vol. 1, 1992), Analysis Methods (Vol. 2, 1994), Practical Use (Vol. 3, 1997).* Monographs in Theoretical Computer Science. Springer-Verlag, 1992-97.
15. Krasner, G. & Pope, S. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. *J.Object-Oriented Programming* 1(3):26-49,1988
16. Kurtenbach, G. & Buxton, W. User Learning and Performance with Marking Menus. In *Proc. Human Factors in Computing Systems*, CHI'94, ACM, 1994, p. 258-264.
17. Kurtenbach, G. and Buxton, W. Issues in Combining Marking and Direct Manipulation Techniques. In *Proc. ACM Symposium on User Interface Software and Technology*, UIST'91, ACM Press, p. 137-144.
18. Kurtenbach, G., Fitzmaurice, G., Baudel, T., Buxton. W. The Design of a GUI Paradigm based on Tablets, Two-hands, and Transparency. In *Proc. ACM Human Factors in Computing Systems*, CHI'97, ACM Press, 1997, p. 35-42.
19. Lehrmann Madsen, O., Møller-Pedersen, B. & Nygaard, K. *Object-Oriented Programming in the Beta Programming Language*, Addison-Wesley, 1993.
20. Linton, M., Vlissides, J., Calder, P. Composing User Interfaces with InterViews. *IEEE Computer*, 22(2):8-22, February 1989.
21. Mackay, W., Ratzer, A. & Janecek, P. Video Artifacts for Design: Bridging the Gap between Abstraction and Detail. In *Proc. ACM Conference on Designing Interactive Systems*, DIS 2000, New York, August 2000, in press.
22. Myers, B. et al. The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Trans. Software Engineering*, 23(6):347-365, June 1997.
23. Myers, B. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs. In *Proc. ACM Symposium on User Interface Software and Technology*, UIST'91, ACM Press, p. 211-220.
24. Myers, B.A. A New Model for Handling Input. *ACM Trans. Information Systems*, 8(3):289-320, 1990.
25. Stone, M., Fishkin, K., Bier, E. The Movable Filter as a User Interface Tool. In *Proc. Human Factors in Computing Systems*, CHI '94, ACM Press, 1994, p. 306-312.
26. Strass, P. IRIS Inventor, a 3D Graphics Toolkit. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA '93, ACM Press, 1993, p. 192-200.
27. Woo, M., Neider, J. & Davis, T. *OpenGL Programming Guide*, Addison-Wesley, 1997.