Translucent Patches
—Dissolving Windows—

Axel Kramer
GMD (German National Research Center for Computer Science)
PO. Box 1316, 53731 Sankt Augustin, Germany
E-mail: axel.kramer@gmd.de

ABSTRACT
This paper presents motivation, design, and algorithms for using and implementing translucent, non-rectangular patches as a substitute for rectangular opaque windows. The underlying metaphor is closer to a mix between the architects yellow paper and the usage of white boards, than to rectangular opaque paper in piles and folders on a desktop.

Translucent patches lead to a unified view of windows, sub-windows and selections, and provide a base from which the tight connection between windows, their content, and applications can be dissolved. It forms one aspect of on-going work to support design activities that involve "marking" media, like paper and white boards, with computers. The central idea of that research is to allow the user to associate structure and meaning dynamically and smoothly to marks on a display surface.

KEYWORDS: Interface metaphors, interaction techniques, irregular shapes, translucency, pen based interfaces.

INTRODUCTION
Historically, window systems came about from the urge to present multiple information contexts at the same time. Indeed, the very first window system presented each window on a separate physical screen. Sutherland took this idea one step further, treating a window as a view onto a large virtual world [38]. In Flex, Kay put multiple windows side by side on the same screen, and finally, within the Smalltalk environment, created the concept of overlapping windows on the same screen [22, 44]. This further developed into the desktop metaphor used in the Xerox Star and subsequently in other commercial products. From then on this kind of presentation of and interaction with information on computers became dominant in the commercial world.

Application-centered User Interfaces
Initially, window systems were application centered. Typically, each window represented one application. The user focused on starting applications in order to do work. Material created in the window of another application could be copied into the current application window, but could only be edited and changed in the window of the originating application. The association between window and application was very tight. Some applications required more than one window, e.g. for iconic command palettes, or for showing various documents of the same type. Some systems allowed windows to be nested: an application window would contain "sub-windows" for open documents of that application [27].

Document-centered User Interfaces
Eventually, the desire for mixing information from different applications into one document became stronger, and the UI-community sought ways of embedding documents of one application into a document of another application in a transparent way, that is, by still allowing the user to edit the embedded document inside the embedding document with the operations and interface of its original application [11, 12, 25]. The perception of a document changed from being of a particular application, to being an assemblage of information from various applications. The document could contain areas and sections of material originating from various applications, and they could all be edited right on the spot, in the same document window. The association between application and window vanished, a window was now associated with a document instead.

Even with embedding documents into each other, the association between the application and the representation itself, that is the characters and graphical objects, is still static. Very few applications can put a different interpretation or meaning on the representation shown to the user. One example that comes to mind are text editors which can turn tab-delineated text into a table, and vice versa. The opportunities to spatially arrange embedded documents is minimal. Typically, the embedded document is restricted to an opaque rectangular area.

Usually, users have to choose an application, or an area within a document, and can only then "fill in the blanks", e.g., they choose a word processor and can then type text, they define an area within a document to be a drawing area and can then draw circles and boxes in that area.

Dynamic Interpretations

While this static association between interpretation and representation might be appropriate for certain domains, we are interested in supporting design activities with computers. In particular, we would like to support design activities that involve marking media, e.g. paper and white boards.

Empirical design studies show that the design process is seldom a straight forward process, proceeding step by step in a linear and logical fashion, but usually a social process of negotiation in which tools provide means of mediating communication on various levels of abstraction and ambiguity. [29, 43]. Koestler points out that a very important component of the creative process is to be able to see the same thing under a different light, from a different point of view [23]. It is no surprise that some of the interesting ideas appear in dreams first; how much more fluid in the manipulation of reality can one get? Arnheim describes the central role that images, shapes, and forms play during the thought process [5].

The interpretation of symbols and gestures used in the design process is determined on the spot, through negotiated understanding, supported by personal experience, the social context, and the common understanding of practitioners in the field. Using computational facilities to support some of the manipulations of symbols in this process comes at a cost. In order to perform operations on symbols, computers require the interpretation of these symbols to be rigid and well defined. Yet, the role of computer tools should not be to constrict the fluidity of the design process nor the fluidity with which meaning is attached to the symbols used in this process. Instead, computer tools should make representation only rigid when necessary for computational operations, at the appropriate level of abstraction chosen by the user.

Thus the central idea of the research this work is embedded in, is to dissolve the static association between representation, its structure, and its interpretation as delineated by a window and its associated application and experiment with a dynamic user driven association between representation, structure and interpretation as the central mechanism for using computers. We will call this approach *representation driven.*

Instead of choosing an application and then "filling in the blanks", this approach puts representation before structure and interpretation. Using a pen, the user draws marks on an electronic display surface, and only when needed, possibly at some later point in time, identifies a subset of the marks and applies some interpretation to it.

Consider an architect in discussion with her client. While talking, listening, and sketching, she jots down numbers in some empty space of her electronic sketchpad. Later, she selects them as a structure and applies a *calculator interpretation* to that structure. While this *calculator interpretation* is applied, new marks in that area will be treated as numbers, or operations particular to the interpretation. When space is really needed for something else the structure can be moved around to some other area. Since it and the other structures are translucent, the drawing below is still visible. In contrast to current user interfaces, the overhead is not paid up-front, but only when the operations are really required by the user.

This paper focuses on aspects of our research as they relate to windowing systems. In particular, we concentrate on concepts and implementation for supporting translucent non-rectangular patches. Patches are dynamically created by the user in order to identify a spatial subset of marks on the display surface. A patch can be associated with an interpretation and thus allows the user to manipulate or transform the identified marks, as well as new marks in particular ways. Translucent patches are uniform structuring tools and unify the concepts of windows, views and selections, and let the user choose to move particular spatial regions to the appropriate level of computational abstraction and rigidity.

The rest of the paper is divided into three sections. The first section discusses related work in the area of supporting design activities, using translucency and irregular shapes in user interfaces. The second section elaborates on the concept of translucent patches. It discusses the usage of translucency and irregular shapes in today's tools. It concludes by presenting interesting operations that do not make sense in traditional application or document based windowing systems, but are interesting for a representation based system. Finally, we present design and algorithms to implement translucent patches.

RELATED WORK

The following overview of related work focuses on the usage of translucency and irregular shapes in user interfaces. Since this paper concentrates on aspects of our research that relate to windowing systems, we will touch the vast body of related work in supporting design activities with computers only briefly.

There are a number of research and application areas that made the support of design activities with computers their focus. One is architectural design and technical drawing. Sketchpad [38], from I.E.Sutherland, is probably the earliest example of such work. He aimed at providing support for creating engineering drawings, and invented a number of concepts commonplace in today's user interfaces. A fixed set of constraints was available to the user for specifying the relationships between graphical objects. The network of graphical objects and constraints then formed a technical drawing. Negroponte and his group did some early work on supporting architects. His approach was governed by an attempt to embed all "intelligence" necessary for the design process into the machine itself [32]. The MediaSpace project [42, 46] investigated the use of video technology to support architectural and engineering design activities across space and time. With the advent of the personal computer, architecture applications and layout programs became commonplace.

Another area of research includes meeting support environments. Early work in this are was done by Doug Engelbarts' group at SRI while doing research on the hypertext system NLS/AUGMENT [15]. Colab is an extension of these ideas, providing personal workstations with private windows for each participant, a large back-projected common viewing area, as well as experimental software tailored to support meetings within this new interactive media [40, 16]. Tivoli [34] added an electronic pen as an input device, and enhanced the meeting support software. Based on a slide metaphor, the electronic pen can be used to enter sketches and lists. Buttons and gestures are provided to perform editing and list operations.

Other approaches blend this technology with video and attempt to support distributed synchronous meetings [19]. Commune [10] and other experiments [9, 30] investigate the impact of sketching facilities when added to distributed design sessions.

A third area of research attempts to formalize design and argument structure. Examples for such systems are NoteCard [18], NLS/AUGMENT [15], and design rationale systems [26].

In the domain of graphical symbols, Fred Lakin presents an approach to support design activities through parsing visual languages [24]. His system operates in a two step process, a free-form sketch entered by the user is parsed against a visual grammar in a second step. Nakagawa argues that lazy recognition provides a better principle for pen interfaces than immediate recognition. [31].

Translucency

The usage of translucency as an aspect in the design of user interfaces is not very common. Some drawing and architectural programs provide (totally-)transparent layers [14]. Adobe Photoshop allows the user to specify an alpha channel associated with an image that can then further be used as a mask in Adobe Premier videos [2].

Staples [39] gives examples and shows mock-ups for extending the flat space of today's user interfaces by the usage of light, space and levels of transparency.

Belge et.al investigate how hard it is for users to discern which object is in which transparent layer. As described in [7] the experiments are based on totally transparent, or opaque slides that exactly overlay. The computer based experiment included tests on access of objects in transparencies only.

Wong provides an interesting observation of an architect using yellow paper [47]. The subsequent design sketch of a computer system is based on a fixed number of exact overlay transparencies that can be named, and easily reordered or turned into opaque transparencies.

An interesting use of translucency is demonstrated in the ClearBoard project [19]. Here, live video images are displayed and overlaid with a translucent sketching area.

The See-Through™ interface [8] provides an innovative idea of applying transparency to command buttons. Translucent command panels can be used to "stamp" commands on underlying objects. Other panels can be used as filters to show the underlying objects in a different way.

Display-PostScript as used in the NeXT display system can associate an alpha channel to bitmaps. Active application of the alpha channel is possible only on the display surface though, and not during printing [1].

None of the papers relate their respective implementation. As can be seen below there are some interesting challenges which make the processing of translucent areas or windows very different from the processing of overlapping opaque display areas.

Irregular Shapes

Most windowing systems today provide primitives that allow programmers to manipulate non rectangular shapes. Probably first embedded into QuickDraw [3], it is also available on Windows [28], and as an X extension [33]. Steinhart [40] shows a possible implementation for regions.

Using irregular shapes in user interfaces has caught on even to a lesser degree than using translucency. A notable exception is the lasso tool, used to manipulate areas in paint programs [4]. Fractal Painter X2 extends this concepts and turns the once short lived lassoed areas into full fledged objects, allowing multiple areas to exist concurrently, and even be saved to the file [17].

Translucent Patches

Our everyday world contains many examples for using translucency. In some technical drawings of three dimensional bodies the foreground surfaces are made translucent, to suggest to the viewer the structure of the hidden surfaces. Rolls of yellow paper are a ubiquitous medium in architects offices. When working on subsequent or alternative versions to a plan, architects often overlay the old plan with a new sheet of yellow paper. Since the old plan is still showing underneath, additions can be sketched without replicating the whole plan. Or particular aspects of the old plan can be traced to be incorporated into the new version.

Annotations done by editors act as translucent layers. Here, the effect of translucency is not created through the use of an actual translucent medium, but by writing the annotations with a pen in handwriting and thus setting them apart from the content written in a typeset font. Both, the annotations and the manuscript are still visible, allowing to relate the annotations to the written material.

Real windows are translucent, letting light in, and letting us see the outside, but providing protection from rain, cold and wind. Very similar to the use of plastic wrapping of certain products. A protective layer, which the customer can see through, yet provides unity of the content and easier handling.

Subtitles in some movies are annoying when not done in a translucent way, that is, if the subtitles are done on a white rectangle covering up the lower part of the movie.

In summary, translucency in our everyday world serves two main purposes. It preserves context, allowing us to relate multiple bodies of information at the same time. It protects content, allowing us to experiment with or manipulate the content without actually effecting it directly.

Non-Rectangular forms and shapes are also a common occurrence in our everyday environment. Natural forms are typically non-rectangular forms. Growing processes produce forms with non-rectangular shapes. For example, plants grow in natural regular, but not rectangular shapes [37]. Maps of old cities reflect the changes to the city layout over time as it follows terrain on the one hand, and a sense of unity on the other [13].

Looking at the content of note-books and white boards, one finds many examples of non-rectangular shapes. Project plans, organizational charts, data models, illustrate the use of spatial information and shapes to communicate information (see figure 1.) [45]. Design of algorithms usually involves the playful simulation of data structures in time. Reflecting and thinking is a growing process that points to the need for spatial layout and structures that are more complex than just rectangular shapes.

Other irregular shapes play together with the usage of translucency. Imagine magazine cover pages with opaque rectangles around article headlines covering up the cover pictures, or annotations of papers and manuscripts, where the annotation is not left transparent in its irregular shape, but are neatly embedded into white rectangles.

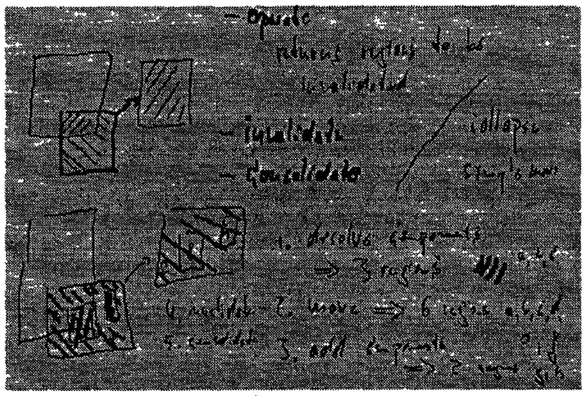


Figure 1. Example for a non-rectangular spatial structures in notebook

For artists shapes define unity and the flow within a painting as perceived by the viewer [20, 36]. Shapes define boundaries of objects perceived and thus set objects apart, or define groupings of objects.

In summary, irregular shapes in our everyday environment arise out of growing processes, and provide a sense of groupness and unity.

The approach presented in this paper employs translucency and irregular shapes as an important infrastructure for selecting, presenting, and manipulating information with computers. In combination with a dynamic, user driven, association between content and interpretation, we attempt to preserve the fluidity necessary during the design process.

User Interaction

Users interact with our system in quite a different way than with common window systems. Translucent patches are created by the user in order to either identify, and thus group, a spatial subset of marks, or to create space. Let's pick up the calculator example from above. An architect and her client are in a discussion about a kitchen extension and work in front of an Architects Electronic Sketchboard as shown in figure 2.

The client brought the plan of his map on disk, and it opened in a translucent patch in the center of the electronic sketchboard, partly covering some work the architect worked on the day before. While discussing the client and the architects both sketched and drew on the sketchboard, extending the original patch by some area to sketch the kitchen extension, sketching a vertical view of the extension in some empty space, and creating a new patch for some comments the architect wrote down.

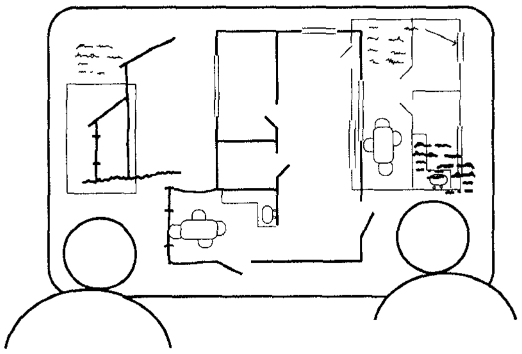


Figure 2. Architects Electronic Sketchboard

The architect knew the client would want a rough cost estimate right away, so she started to write down the main cost items during the discussion, see figure 3.

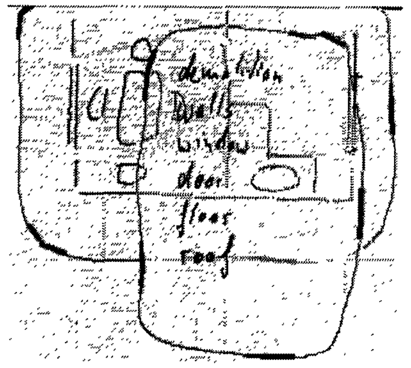
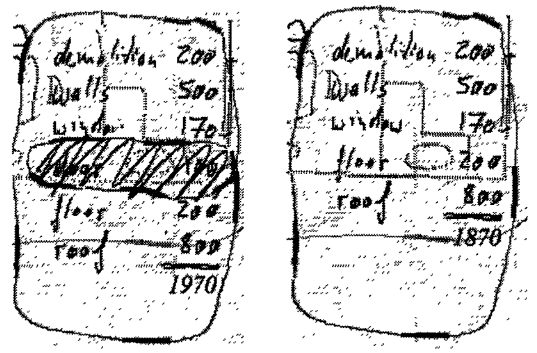


Figure 3. Architects scribbles in translucent patch

Now, at the end of the session, she writes a number next to each item, a double bar below the list of numbers, and then chooses to apply a *calculator interpretation* to the patch. The interpretation recognizes the list of numbers, and adds them up below the double bar.



Figures 4. Estimates in calculator interpretation

The client decides that he might do without the additional door. The architect uses a gesture to create a patch including the door item, and some gesture to delete the patch with content. The door item in the list is deleted, and, due to the *calculator interpretation*, elements below are shifted up, and the sum is re-calculated (see figure 4). Since the patch is translucent, the other plan below is still visible. It can be moved to the top by clicking on one of its handles, one of the thicker areas around the border.

Untypical Operations

Operations on patches have a different flavor than operations on windows in regular window systems. Some are similar, like bringing a patch to the top, moving it around, or collapsing it. The untypical operations relate to the disassociation of windows from their content.

Creating patches

A patch is created by the user with a gestural mark selecting either empty space, or some subset of marks in some other patch. The interpretation associated with a patch decides what gestural mark creates a new patch, for example, where the default gesture is a closed shape above a minimal size, a *list-interpretation* could provide a gesture for enclosing a list entry with a patch by a vertical bar gesture, or a *text interpretation* could enclose a word in a patch after it has been double clicked.

Patches essentially act like selections and subviews, in that they are used to create structures, spatial subsets of marks, and in that operations are provided that act on the selected subset as a whole.

As a default, a new patch is contained in the patch it was created in. Typically, a new patch lifts up the spatial subset of content marks and owns them from then on.

Deleting, dissolving, and clearing a patch

There are three operations that relate to deleting a patch, or its content. The user might decide to delete the patch and its content. Or the user might just want to dissolve the patch as a unit of marks, thus essentially moving the marks contained in the patch to the containing patch below, and then deleting the patch. A third possible operation is to clear the inside of a patch, leaving the patch itself intact.

Enlarging patches

The area a patch includes may be enlarged or made smaller to include or exclude space or marks. This is done via gestures that lead out of the patch into the space and back into the patch (for enlarging), or from the outside inside, and back into the outside for making it smaller. Some interpretations might grow or shrink the patch automatically when content is added or removed.

Assigning interpretation

Each patch is at least associated with the *basic interpretation.* This interpretation interprets marks as plain ink, provides some gestures for deleting marks, and gestures for creating new patches and assigning a different interpretation. When a patch is created within a particular interpretation, the interpretation might decide to assign some other interpretation than the basic interpretation to the new patch. For example, when the user assigned a *textual interpretation* to a patch, newly created patches inside that patch might be initialized with the *text interpretation* also.

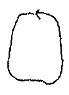
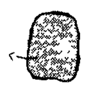
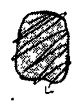
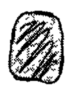
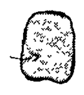
Lifting and releasing content

The user might want to lift marks from underlying patches into a patch above, or release some subsets of marks to the containing patch.

Pearls

Drawing a gesture from the outside into a patch generates a pearl, a small circular object, at the beginning of the gesture. A pearl is a handle to the patch, and can be used to collapse and expand it, as well as to change its relative positions to patches above and below. Pearls can be moved around, and they can be embedded in a patch, like any other object on the display surface. The user can write a label, or a little sketch next to a pearl, or just arrange a number of pearls in a particular spatial arrangement.

We are still experimenting with a consistent set of gestures for these operations. The challenge is not only to make the gestures consistent and memorable, but also to preserve the fluidity of interaction. Table 1. lists the patch-related gestures we are using for now. The little arrow heads are not part of the gesture, they just visualize the drawing direction.

create	dissolve	delete	clear	pearl
				
draw closed polygon	straight gesture out	zig zag crossing border	zig zag filling patch	line into patch

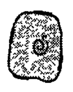
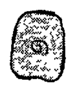
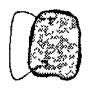
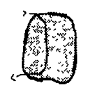
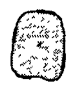
lift	release	grow	shrink	assign
				
counter-clockwise spiral	clockwise spiral	inside to outside to inside	outside to inside to outside	small circle -> pop up

Table 1. Current patch related gestures

The current deletion gesture is an example of a gesture that is memorable and consistent with other deletion gestures, yet feels not swift enough in actual execution.

Gesture marks are differentiated from content marks by the pressure of the pen or an additional button on the pen when no pressure levels available. This proves very useful in practice, but needs to be evaluated and compared to traditional time-out mechanisms.

Some gestures execute immediately (e.g. create patch, dissolve), others can be executed at some later time by tapping them (e.g. delete). Non-yet executed gestures act as if they are regular components, e.g. they are lifted like other components into a new patch. Besides executing them with a tap, the user can erase them with a zig-zag mark (essentially a meta-gesture).

ALGORITHMS

Implementing translucent patches is quite different from implementing a window system for opaque rectangular windows. Although there are similarities: e.g. one deals with overlapping surfaces that can be nested and contain a variety of graphical objects, mostly ink, but also text, circles, lines, etc., there is a main difference in that covered patches still need to display the content of the covered areas, albeit with a different intensity.

```
Class Region    instance creation
        empty "return new region"
        fromRectangle: r "return region based on rectangle"
        fromPolygon: p "return region based on polygon"

    operations
        add: aRegion    "add given region to receiver"
        subtract: aRegion "subtract given region from receiver"
        intersect: aRegion "return the intersection between
                                    receiver and argument"
    testing
        containsPoint: p "return true if receiver contains point"
```

Figure 5. Class Region

The simple minded approach to overlapping opaque windows is to just paint all the windows from the bottom up, and expect that windows on the top will cover windows further down. The analogy for transparent windows would be to paint the windows from the bottom up, but "massage", e.g. lighten up the surface below a window, before it is drawn, and to not draw over the surface with the background color as the first invalidation operation.

This conceptual approach could be implemented well if one would deal mostly with pixmaps. One would need to traverse every pixel and change its color value according to the translucency of the window above. This approach has several disadvantages:

- background patterns would eventually add up to a muddy gray. This is closer to the physical analogy, but decreases the possible contrast between background and foreground, which we would like to avoid (compare Kandinsky's watercolor: Träumerische Regung [21]),
- covered patches would not have a choice to not display certain information if covered , e.g. certain information might be so detailed that rendering does not make sense. One might not want to draw control symbols when covered,
- and the opposite: covering patches would not have a choice of asking covered patches to represent themselves in a particular way, e.g. as in see-through interfaces [8],
- does not work very well for current drawing models that developed away from bit mapped graphics towards graphical primitives, e.g. PostScript printing would not be supported easily.

Instead, the approach presented in this paper keeps track of the appropriate intensity values for each area of a patch that is overlapped. The intensity value reflects the amount of light that still comes through the patches above. An intensity value of 1.0 denotes no change in brightness, whereas an intensity value of 0.0 is used for regions that are covered by an opaque patch. When rendering the patch content, color values will be changed according to the intensity value.

Regions

An important infrastructure that is used in the design and implementation of this algorithm is the notion of a region, a spatial, two-dimensional area, that supports particular region operations, and that can later be used as a clipping-region when rendering graphics. The definition in figure 5., shows an excerpt of typical operations that are relevant in the context of this paper.

Regions can typically be created empty, based on a rectangle, or based on a polygon. Interesting operations include adding a region to another region, subtracting a region from another region, and computing a region that contains the intersection between to regions, that is the overlapping region. One can test if a point is contained in a given region, and one can use a region as the clipping region when drawing objects. Note, if one wants to experiment with translucent patches using the given algorithms, but not incur the space and time overhead associated with regular regions, one can use rectangular regions instead (which are much easier to implement and represent), and restrict all patches to have a rectangular outline.

The difference between a patch and a region is that a region defines some shape in a two dimensional coordinate system. A patch is a display object that contains other graphical components, an associated translucency value (how much light shines through), an interpretation, and an outline.

Two Observations

There are two interesting observations that have impact on algorithms for translucent patches. The first relates to the association between regions and patches, and the second adds an additional constraint by showing the non-local properties translucent patches have with respect to the patches they cover.

Space Filling Perspective

The first observation is to project all intersections between patches on one plane, that is flatten them out, and realize that they represent the whole without any overlapping. Figure 6. illustrates this. The patches A, B, C, and Z overlap. Z is below A is below C is below B. The right figure shows the intersecting regions. The sum of all regions a, b, c, d, e, f, g, z cover the whole surface.

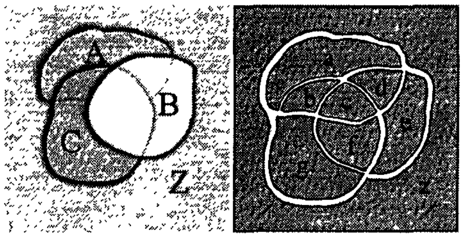


Figure 6: Translucent Patches, Intersecting Regions

From the patches point of view, each has a collection of regions with associated intensity values that cover the patches area. For example, the patch A has four regions a, b, c, d with respective intensity values: 1, 1/2, 1/4, 1/2 (assuming that each patch has the same translucency value).

From the regions point of view, each has a sequence of patches associated with it, from the uncovered patch to the patch covered by all other patches in this region. For example, the region c covers the patches: B, C, A, Z. The intensity value can be implicit by the position in the sequence, or computed dynamically by adding up the translucency values for each patch in the sequence.

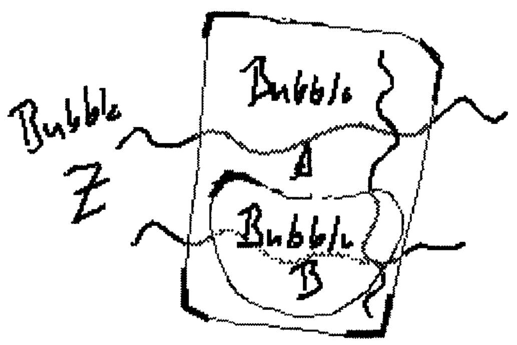


Figure 7. Nested Translucent patches

Non Local Rendering

An important consideration for computing the appropriate intersecting regions is the observation that components inside a patch have impact on the rendering of the patches below. Figure 7. illustrates this problem.

A patch A contains a patch B (that is patch B is a component of A, e.g. if A moves B moves). The one wavy vertical line is a component of patch A. The two wavy horizontal lines in the background are components of a patch Z below. Notice, that the lower one is displayed with less intensity while being below patch B. That is, the region of a nested translucent component needs to be treated as a "first-class" translucent region, so that all objects below appear with the correct intensity.

This is very different from opaque windows, where the components an opaque window contains have no impact on the rendering of components of windows below.

Basic Data Structures

The following operations are based on the region-centered view. A previous system was implemented in the patch centered view, but due to the non-local rendering properties, the implementation was more complex and resource demanding. DisplayRegions are like regions, but each contains in addition to the regular region state a list of patches that totally cover that region. The regions are managed centrally for a given display surface, or in our case, the window in which the transparent patches live.

One object, a RegionManager, contains the list of regions that totally cover the display, and also implements behavior for adding, dissolving, reshaping, and moving regions and patches.

Fundamental Functions

There are two fundamental functions which are used to implement a variety of operations on patches. One adds a region to a patch, the other removes a region from a patch.

Adding Region to Patch

Adding a region to a patch essentially enlarges the patch by the given region. This is done by enumerating over the existing regions and computing the intersection of the new region with an existing region.

If there is an intersection, the intersection becomes a region that includes the patches the existing region referred to, as well as the patch the region is being added too. If the intersection covers all of the region to be added, the algorithm has come to an end. If the existing region is covered totally by the intersection, the existing region is substituted by the new intersection and will be removed. If the new region still contains some area after all the existing regions have been enumerated, we have a new region that is only displaying the given patch. All intersections will be DisplayRegions that have the patch at the end of their patch list. All regions that have changed and should be invalidated eventually will be returned.

```
addRegion: region for: patch
    "Add the given region for the patch to the list of regions  Split regions if
necessary. Note, the region might be changed, or even released after this algorithm.
Return the display regions that got added and should be invalidated eventually."

    | o i end new dead |
    o := region. i  = DisplayRegion empty. end := false.
    new := OrderedCollection new. dead := OrderedCollection new.
    self do:[:r|
        (i becomesRegion. o intersecting. r) == nil ifFalse:[
            i patches: (r patches copyWith: patch).
            (o minus: i) == nil ifTrue:[end := true  o release].
            (r minus: i) == nil ifTrue:[dead add  r. r release]
            new add: i.
            end ifTrue:[self removeAll. dead. self addAll: new  ^new].
            i := DisplayRegion empty]].
    i release
    self removeAll: dead.
    new add: (o asDisplayRegionFor: patch). self addAll: new.
    ^new
```

Figure 8. Adding a region to a patch

Removing Region from Patch

The other essential function is to remove a region from a given patch. The algorithm is very similar to the algorithm for adding a region. The main difference is that the given patch is removed when the region intersects with an existing region. The algorithm ends when the argument region is shrunken to an empty region, or if all of the regions referring to the patch have been enumerated. If a region is given as the argument that is actually larger than sum of regions covering the patch, the left over region will be released. As in the adding algorithm, regions that were changed, are returned at the end.

Both functions, adding and removing a region from a patch, can potentially leave regions behind that refer to the same collection of patches. These can be consolidated to save resources and processing time.

Patch Functions

Operations like adding, deleting, reshaping and moving a patch can be implemented on top of the two fundamental functions given above.

Adding Patch

Adding a patch requires computing a region for the patch and then calling the method above with the new region and the patch. An interpretation associated with the containing patch will create a patch with an outline extracted from the gesture. It will then transfer components of the containing patch intersecting the new patch over to the new patch, add the patch with its region, and then invalidate the returned changed regions.

Dissolving Patch

Dissolving a patch is not based on one of the fundamental operations. Instead, all existing regions are enumerated, and for all that refer to the given patch, the patch is removed from the patch list of the display region. If there is no patch left in the patch list of the display region, the display region will be removed and released. Otherwise the region needs to be returned, so it can be invalidated later.

Growing and Shrinking

Growing is implemented by computing the region that needs to be added, and then calling the adding function. Shrinking is done analogously with the remove region function.

Reshaping Patch

Reshaping a patch with a new outline, involves a combination of adding and removing a region for a patch. First, a new region, based on the new outline is computed. Subtracting the new region from the existing outline of the patch results in a region of the patch that needs to be deleted. Subtracting the existing outline from the new region results in a region that needs to be added to the patch.

Moving Patch

Moving a patch is just computing its new outline, and reshaping the patch with it. The regions returned as changed must include all the display regions referring to the patch. An interpretation will also need to move all components and invalidate returned regions. Attention needs to be paid to patches contained in the patch being moved. These need to be removed first, before moving the patch and are added back in afterwards. This complication is due to the impact local patches have on global rendering.

```
removeRegion: region for. patch
    "Remove the given region for the patchfrom the list of regions. Split regions if
necessary. Note, the region r will be released in this algorithm. Return the list of
regions that changed, and thus should be invalidated later."

    | o i end patches new |
    o := region. i := DisplayRegion empty  end := false.
    new := OrderedCollection new
    (self regionsFor. patch) do:[.r|
        (i becomesRegion: o intersecting: r) == nil ifFalse [
            (o minus: i) == nil ifTrue:[end := true].
            patches := r patches.
            (r minus: i) == nil ifTrue:["r totally covered"
                (self remove. r) release].
            patches size > 1 ifTrue:[
                i patches: (patches copyWithout. patch).
                self addLast: (new add: i)
                i := DisplayRegion empty].
        end ifTrue.[i release. o release  ^new]]].
    i release. o release
    ^new
```

Figure 9. Removing a region from a patch

Raising and Lowering

Raising and lowering patches is done by changing the patch sequence for the display regions that contain a reference to the patch.

Invalidation

For now the intensity value, that is, the value which determines by how much a color value needs to change, because it is covered by some patch, is computed as a function of the number of patches in a display region. It would be easy to change, for example to give each patch a particular translucency value and change translucency values with gestures (e.g. to make a patch opaque). Invalidation asks the containing window for a renderer, and then enumerates through the display regions. The clipping region is set based on the display region, the region is filled with the background color of the top patch, and then each patch in the display region sequence is asked to render itself, after the intensity value has been set. The renderer will transform each request for a color value according to the intensity value set.

Implementation

The above algorithms are embedded in a system that has been implemented in Smalltalk-80, VW 4.1, using Windows 3.1 on a 486/66 with a Wacom pressure sensitive digitizer, and Windows for Pen Computing on a Compaq Concerto. The pre-predecessor of this system was implemented in the summer of 1987/88 while visiting the University of Canterbury, Christchurch, New Zealand. At the time the system provided a mechanism for defining gestures in a declarative manner. The predecessor of this system, without translucent patches but with user defined structures, was created in the spring of 1992 and was used to create slides for a talk and a demonstration video. At the time of this writing the framework for integrating dynamic interpretations has been designed and is in early stages of implementation.

SUMMARY AND FUTURE WORK

This paper presented one aspect of our current research, as it relates to windowing systems. Instead of rectangular opaque windows, we use translucent patches as a unifying mechanism that subsumes windows, sub views and selections. Instead of maintaining a static association between a window and its content, we attempt to make that association dynamic and user driven, in order to support design activities that require a fluid manipulation of symbols on a marking media like paper and whiteboard. The central idea is to let the user choose the level of rigidity groups of symbols have, so they can be processed by computer programs.

Using patches that way leads to interesting operations that are not common in regular window systems, the assignment of structure to marks on the surface is more dynamic. We presented the current set of gestures that we implemented to interact with patches.

Our implementation of translucent patches is based on two insights: (1) the flattening of the intersections of the patches covers the whole space and provides two points of views for associating patches with intersecting regions, (2) nested patches effect the rendering of non-local components. The algorithms presented could also be used in a less resource demanding environment that limits translucent patches to be rectangular.

Lots of work still needs to be done, both in the area of implementation and user testing. Our set of gestures is still under change. There is a tension between getting a coherent set of gestures that still allows a fluid and efficient interaction with the system. We plan to do more than our current informal user testing once the set of gestures stabilize. Our differentiation of gestures and content marks works very well in practice and needs to be compared against time-out mechanisms and modes, as used in commercial pen-systems.

Associating interpretations with patches is in the early stages of implementation. Our goal is to provide a framework in which we can embed various kinds of interpretations easily. Creating and implementing interpretations on a suitable level of abstraction is an interesting challenge.

ACKNOWLEDGMENTS

I like to thank Thomas Christaller and Reinhard Keil-Slawik for valuable discussions and comments. Peter Wisskirchen for enabling me to do this work in the context of the GMD. Andreas Genau, Diana Merry-Shapiro, and Markus Sohlenkamp for reviewing drafts of the paper.

BIBLIOGRAPHY

1. Adobe Systems Inc., *The Display PostScript System Reference Manual*, Adobe Systems Inc., 1990
2. Adobe Systems Inc., *Photoshop and Premiere Users Manuals*, Adobe Systems Inc., 1993
3. Apple Inc., *Inside Macintosh*, Addison-Wesley, New York, 1984
4. Apple Inc., *MacPaint Users Manual*, Apple, 1984?
5. Arnheim, R., *Visual Thinking*, University of California Press, 1969
6. Bekker, M., *Representational Issues Related to Communication in Design Teams*, InterCHI 93, Adjunct Proceedings, 1993
7. Belge, M. et al., *Back to the Future: A Graphical Layering System inspired by Transparent Paper*, INTERCHI 93 Adjunct Proc., 1993
8. Bier, E. et al, *Toolglass and Magic Lenses: The See-Through Interface*, ACM Computer Graphics Proceedings, 1993
9. Bly, S., *A Use of Drawing Surfaces in Different Collaborative Settings*, CSCW 88, Portland, 1988
10. Bly, S. and Minneman, S., *Commune: A Shared Drawing Surface*, in SIGOIS Bulletin, 1990
11. Brockschmidt, *Inside OLE 2*, Microsoft Press, 1994

12. Carr, R. and Shafer, D., The Power of PenPoint, Addison-Wesley, 1991

13. Ching, F., *Architecture: Form Space & Order*, Van Nostrand Reinhold, New York, 1979

14. Claris, *MacDraw-Pro Users Manual*, Claris, 1991

15. Engelbart, D., *The Augmented Knowledge Workshop*, A History of Personal Workstations, Ed. A. Goldberg, Addison Wesley, 1988

16. Foster, G. and Tatar, D., *Experiments in Computer Support for Teamwork - Colab*, Video, Xerox PARC, 1988

17. Fractal Design, *FractalPainter X2 Users Manual*, Fractal Design, 1994

18. Halasz, F., *Reflections on NoteCards: Seven Issues for the Next Generation of Hypertext Systems*, in: CACM 31, 7 (1988), 836-851

19. Ishii, H. and Arita, K., *ClearFace: Translucent Multiuser Interface for TeamWorkStations*, Proceedings of the Second European Conference on Computer Supported Cooperative Work, 1991

20. Kandinsky, *Point and Line to Plane*, Dover, New York, 1979

21. Kandinsky, *Watercolors by Kandinsky at the Guggenheim Museum*, Guggenheim Museum, New York, 1991

22. Kay, A., *The Early History of Smalltalk*, HOPL-II, ACM, 1993

23. Koestler, A., *The Act of Creation*, Penguin, 1964

24. Lakin, F., Parsing Visual Languages, Visual Languages, Eds. S.Chang, T.Ichikawa, P.Ligomendides, Plenum Publishing, 1986

25. Linton, M., *Fresco-Slides*, Stanford InterViews FTP Server, 1993

26. MacLean, A., et.al, *Design Rationale: The Argument behind the Artifact*, CHI 89, ACM, 1989

27. Microsoft Corporation, *Microsoft Windows 3.1 Users Guide*, Microsoft Corporation, 1992

28. Microsoft Corporation, *Microsoft Windows 3.1 SDK*, Microsoft Corporation, 1992

29. Minneman, S., *The Social Construction of a Technical Reality: Empirical Studies of Group Engineering Design Practice*, Dissertation, Xerox PARC, SSL-91-22, 1991

30. Minneman, S. and Bly, S., *Managing a Trois: A Study of a Multi-User Drawing Tool in Distributed Design Work*, Xerox PARC, SSL-91-118

31. Nakagawa, M. et.al., *Lazy Recognition as a Principle of Pen Interfaces*, InterCHI 93 Adjunct Proceedings, 1993

32. Negroponte, N., *Soft Architecture Machines*, MIT Press, 1975

33. Packard, K., *X11-Nonrectangular Window Shape Extensions*, MIT-Consortium, 1989

34. Pedersen E. et.al., *Tivoli: An Elctronic Whiteboard for Informal Workgroup Meetings*, InterCHI 93 Conference Proceedings, 1993

35. Perlin, K. and Fox, D., *Pad: An Alternative Approach to the Computer Interface*, ACM Computer Graphics Proceedings, 1993

36. Peters, Joye, *Using Shapes in Paintings*, Private Communications, 1994

37. Prusinkiewicz, P. and Lindenmeyer, A. *The Algorithmic Beauty of Plants*, Springer Verlag, New York, 1990

38. Sutherland, I.E., *Sketchpad-A Man Machine Graphical Communication System*, MIT Lincoln Laboratory Techincal Report no. 296, 1963

39. Staples, L, *Representation in Virtual Space: Visual Convention in the Graphical User Interface*, INTERCHI 93, ACM, 1993

40. Stefik, M. et al., *Beyond the Chalkboard: Computer support for collaboration and problem solving in meetings*, CACM, 30(1), 1987

41. Steinhart, J.E., *Scanline Coherent Shape Algebra*, in: Graphics Gems II, J.Arvo (ed.), Academic Press, San Diego, 1991

42. Stults, B., *Experimental Uses of Video to Support Design Activities*, Xerox PARC, 1988

43. Tang, J.C., *Listing, Drawing, and Gesturing in Design: A Study of the Use of Shared Workspaces by Design Teams*, Dissertation, Xerox PARC, SSL-89-3, 1989

44. Teitelman, W., *Ten Years of Window Systems - A Retrospective View, Methodology of Window Management*, Eds. Hopgood et.al, Springer, New York, 1986

45. Tufte, E.R., *Envisioning Information*, Graphics Press, 1990

46. Weber, K. and Minneman, S., *The Office Design Project*, Video, Xerox PARC, 1987

47. Wong, Y.Y, *Layer Tool: Support for Progressive Design*, InterCHI 93 Adjunct Proceedings, 1993