Using Pixel Rewrites for Shape-Rich Interaction

George W. Furnas **Yan Qu**
School of Information
University of Michigan
Ann Arbor, MI
+1 734 763-0076
{ furnas, yqu } @ umich.edu

ABSTRACT

This paper introduces new interactive ways to create, manipulate and analyze shapes, even when those shapes do not have simple algebraic generators. This is made possible by using pixel-pattern rewrites to compute directly with bitmap representations. Such rewrites also permit the definition of *functionality maps,* bitmaps that specify the spatial scope of application functionality, and *organic-widgets*, implemented right in the pixels to have arbitrary form, integrated with the shape needs of the applications. Together these features should increase our capabilities for working with rich spatial domains.

KEYWORDS:

Graphical rewrites, pixel rewrites, graphical interaction, shape manipulation, shape analysis.

INTRODUCTION

Many shapes in nature have no simple analytic description: the ragged boundaries of a marsh, the meandering and branching path of a river, the subtle outline of a thighbone. For advanced analysis or computer manipulation, scientists typically convert images of such shapes, by a process called "vectorization," into complex models in analytic geometry. Similarly, designers work predominantly with algebraic, "vector-based" models of CAD systems, and as a result it can be a struggle to create more naturalistic shapes (e.g., an artificial thighbone, a new car-body-shape).

The work here is part of a research program to explore the possibilities offered by an alternative computational approach, one working directly with bitmap representations of shape. The basic inspiration is that bitmaps can easily capture shapes that are difficult for algebraic expressions – a digital photograph easily captures the shape of the marsh, river, or thighbone outline. New pixel-based shape manipulation algorithms allow increasingly powerful computation directly with these bitmap shapes, opening the way for their more serious use.

Making such pixel-based computations available for human interaction in shape-rich tasks is, however, a research issue in itself. In this paper, following preliminary work by [7], we use accumulating experience with basic pixel-rewrite algorithms to provide new interactive ways to create, manipulate and analyze bitmap shapes, and new ways for users to control that functionality. The remainder of this section gives a brief introduction to pixel rewriting, and a survey of related work. The second section shows how new pixel rewriting algorithms for computation with shapes can be coupled to user input to provide new, pixel-based interactive application functionality. The third section shows how pixel manipulations allow for ways to control application functionality, using *functionality bitmaps* and *organic-widgets*. The conclusion gives a summary and suggests future work.

Pixel Rewriting Basics

To work with pixellated representations of shape we use a Pixel Rewrite System (PRS) (e.g., [5]). A PRS is essentially a graphical version of a rule-based production system from Artificial Intelligence [3]: rules match specified local pixel patterns on a canvas or "field" and then rewrite them as different patterns. As carefully crafted sets of rules repeatedly match and rewrite, the content of the pixel field evolves over time, and shapes and contours change in ways contingent on their contexts. The result is a type of versatile pixel computation that can work with arbitrary pixel shapes without ever having to convert them to algebraic representations.

Figure 1 shows some simple rules, building blocks often used in complex rule sets. The rule in (a) we call a "Flood" since it will cause any pink color to spread out and take over any neighboring white. It does this one pixel at a time, by rewriting any white neighbor of a pink pixel to be also pink. (This is a pixel-rewrite analogue of "filling" in "Paint" programs.) A rule like this can, for example, tell if two things are connected – start the flood at one and see if it reaches the other. That capability is used, for example to identify the full extent of a "blob" to be manipulated (everything the flood can reach), or to send a "signal" from one part of a shape to other connected parts.

Control flow in a pixel rewrite system is accomplished using conflict resolution: When there are multiple matches, a "winner" is selected based on factors like explicit rule priorities and recency of match. For example, resolving conflicts based on "other things being equal, fire the oldest existing match" is what allows Flood (a) to spread out breadth-first.

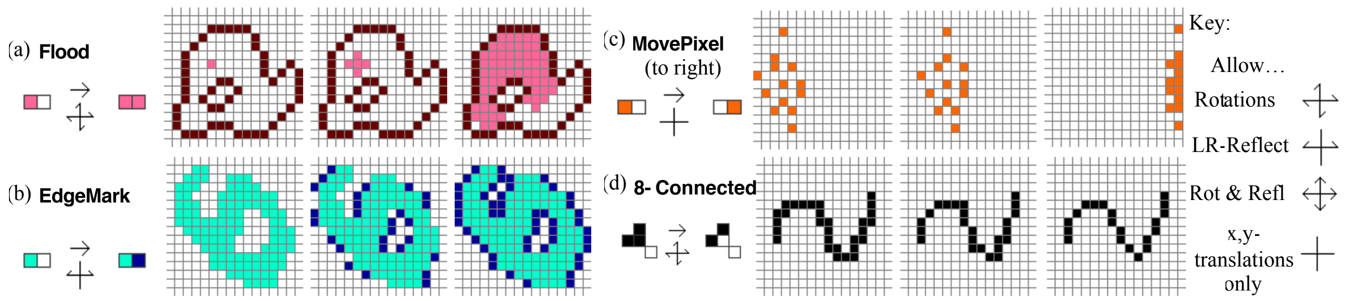

Figure 1: Four single-rule systems. The rules can match rotated or reflected versions of their patterns if so indicated.

Other rules can be used to mark special visual attributes of the bitmap for further processing. One example, in (b), is EdgeMark, which marks with blue all Left and Right edges where turquoise meets gray. Still other rules provide the basics of moving things around, as shown in (c) where the rule simply moves orange pixels to the right. Yet other rules explicitly change the shapes of things. For example, the rule in (d) removes the square corners from curves.

Furnas, et al. [5][6] have shown how these and similar simple rules can be used in many ways, as building blocks of more complicated algorithms for working with shape. For example, a monochromatic "blob" of arbitrary size and shape can be moved over one pixel quite simply with only local rewrites. First, it is flooded to identify its total extent. Then the external leading edge is marked for extension and the interior trailing edge marked in a different color for deletion. A rule then deletes the marked trailing edge pixels, and another converts the marked leading edge to the original blob color. The result is a blob of the original shape moved over one pixel. (We can move multicolor blobs, using an underlying layer that has a monochrome "shadow" of the blob shape to be moved.)

The ability to move things around is a fundamental piece of functionality for interacting with shapes. The fact that we can do it right in the pixels means that we can move arbitrary blobs, even if they do not have easy vectorized representations. If they are in the pixels we can move them. It also means that we can make variations of the algorithms that take into account local pixel context – resulting in versions of blob-move that react to context – for example avoiding obstacles, deforming around them, recursively pushing them out of the way, as desired.

The primary interest here is in those algorithms that provide functionality valuable for end user interactive applications. For example, [6] presents a pixel-rewrite algorithm to find the shortest paths between arbitrarily shaped objects through arbitrarily shaped obstacle fields. This can be used directly to automatically draw useful curves for users, say to connect two designated components in an engineering layout diagram. The algorithms can also build on each other. For example, suppose a user draws some such connecting curve by hand and then decides she has drawn it to be too long and loose, and wants to "tighten it up". If she had drawn the curve right in the pixels, she previously would have had to do a lot of erasing and redrawing. Instead we can use the shortest-path algorithm as a component in a larger algorithm that can successively shorten such long pixel curves, right in the pixel array. (A simple example of such shortening is in Figure 2. See [5] for the algorithm.)

As another example, it is common in graphical design to want to move groups of things closer together, or to spread them out evenly over arbitrarily shaped areas. Typically, such positioning is done by hand. We can do this automatically with a PRS by combining two basic algorithms. First is a "pixel-radar" algorithm that starts at some location and can tell in what direction the nearest object or obstacle lies. It sends out a radiating "radar" signal via a flood which, upon hitting another object, initiates a reverse flood, or "echo". The reflected echo eventually reaches the original source, and its direction of arrival can be used by subsequent algorithms. For example, combining it with the blob-move algorithm, we can get a blob to re-position itself closer to the reflecting object. Running this iteratively in parallel for a set of blobs, we create an "auto-converge-by-radar" capability, causing the arbitrary shapes to cluster together ever more closely. Alternately, we can have the blob move away from the reflection signal, producing an "auto-disperse-by-radar" capability, and it will eventually spread the arbitrarily shaped objects out in surrounding arbitrarily shaped areas, automatically.

Note how the pixel level computations of these algorithms differ from those used in early stages of computer vision, e.g., in morphological analysis [12], or in image processing [11] and photo manipulation (e.g., Photoshop®). In those systems, typically a single "rule" applies simultaneously at all locations to transform quantitatively (e.g., with Gaussian filters) all the pixel values in the image as a function of their neighbors. These accomplish operations like blurring, edge enhancement or repairs of small gaps. In a PRS, the complexly self-sequencing sets of rules, each of which has rich non-linear dependence on local context, accomplish more qualitatively complex behavior, including moving things around,

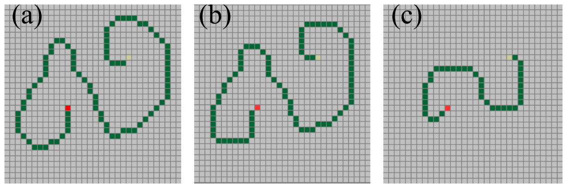

Figure 2: "Tightening up" a curve. The endpoints are fixed, but the curve (a) is one iteration shorter in (b), and several more iterations shorter in (c).

analyzing geometric properties, and finding routes.

The rewrites used in PRSs are technically most closely related to formal systems called Array Grammars (AG)[13], that rewrite elements in a two- or higher-dimensional grid. In most AG work, like other grammar-based systems for shape (e.g., L-systems [9] and Shape Grammars[15]), the primary interest has been on parsing and generating shapes, and as a result work has focused on the highly tractable but less powerful Context Free (CF) or Context Sensitive (CS) grammars – systems that make very restricted use of context in the match and rewrite process. This rather technical point is important because the unrestricted rewrites we use in our PRSs are formally known to be much more powerful – they can implement any computable function. Such power, enhanced by a so-called conflict resolution scheme (ways for the programmer to control what happens when two different rules want to fire at once) allows us a versatile way to manipulate and transform, in quite arbitrary ways, shape on the canvas.

Several of these sorts of powerful grid-based rewrite systems have been explored in the HCI community. Furnas [4] proposed BITPICT, a pixel rewrite system, as a possible model for a kind of forward chaining graphical reasoning. Yamamoto [16] devised several powerful programming extensions to the BITPICT language. In neither of these was there any special focus on the potential for interacting with rich shapes.

Several other systems (e.g., Agentsheets[10], Chemtrains [1], and Coco/Kidsim[14]) have pursued grid-based rewrites with an emphasis on the ability to make simulations for teaching programming or fostering scientific understanding of processes. Though they explored various technical complexities along the way that are useful to us (on conflict resolution, modularization, and control), many aspects of these systems (e.g., large, many-pixel cell sizes; non-pixel data structures) were designed to support the simulation and teaching aims. Here by contrast, we are interested in manipulating and analyzing rich shapes. As a result, the system is optimized to work with individual pixels, bitmaps are the main data structure throughout, and rule sets of interest here are for useful shape manipulations, not simulation.

The pixel rewrite process at the core of a PRS amounts to a kind of graphical search and replace capability – each rule "searches" for a match to its left-hand side and "replaces" the match with the rule's indicated right-hand side. Understood this way, it is useful to compare a PRS to the powerful graphical search and replace capability Kurlander&Bier [8] developed for interactive graphical editing. Users could make global changes to graphical objects, such as changing all red rectangles to green ovals. This differs in two important ways from the work here. First, they focused on changing attributes of "draw"-type vector graphical objects (like red rectangles). We are interested in complex manipulation of raster shapes, so we work directly with the pixels of the raster array. Second, in their system a single pass of the search and replace operation was conceived of as the user's desired unit task. ("Go make that one type of change and get back to me.") In contrast, here we use whole sets of rules, pre-authored by an application designer, that run in complicated sequences to perform graphical computations. The complex graphical manipulations that result are the unit task for the user ("Go analyze or transform this configuration [which happens to be accomplished with tens or thousands of diverse pixel rewrites] and then get back to me.") Kurlander and Bier mentioned that they could work with pixels, and even could do recursive calling, to create fractals, for example, but the rich pixel computation for shape manipulation was not the focus of their exploration.

Finally, viewers often find the grid computations of a PRS reminiscent of the computation seen in Cellular Automata (CA)[2]. In a CA, cells in a grid change state as a function of the states of their neighbors, causing a starting configuration of cells to evolve over time. Such systems help scientists explore the emergent consequences of various neighborhood-dependent transition rules. Classical CAs are, however, often considered hard to program to achieve *desired* (as opposed to explore emergent) results – a critical handicap if the goal is to provide some particular useful shape manipulation capability to a user. Since the patterns in rules of PRSs are often explicitly relevant to the desired resulting functionality, we find them easier to program than CAs, and therefore well suited for giving users specific shape-interaction capabilities.

USER INTERACTION 1 – NEW END-USER-APPLICATION FUNCTIONALITY

The most basic, important impact of the new PRS algorithms for shape manipulation is that the new functionality can be made available, in a straightforward way, to end users. For example, it has not been possible before for users to draw freehand scribbles and then fix them up without having to erase and redraw. Simply by invoking the algorithms on an indicated input, users can smooth portions of their scribbles, or globally shorten ("tighten up") their pixel-curves.

The algorithms of [5] and [6] easily suggest end-usable functionality for *manipulation* of shape. Other algorithms allow the interactive *analysis* of shapes as well. For example, imagine a species of exotic snail has invaded a watershed in the Lake District (Figure 3). An ecologist assessing the threat wants to know which other water system is nearest. Working directly with a pixel map of the area, she clicks on the infected lake, launching a process that basically floods out from the original infected lake-system to find the first uninfected one it hits. Rewrites first "flood" the original water system from her seed click-point (turning it red). Then a second, "search"-flood (yellow) begins from the boundary of the infected lake, spreading until it first touches another water-colored pixel. The touch is identified by a high priority rule, which then marks the location of the touch (reddish orange). The mark acts as a seed for another flood which recolors the touched watershed. The recolored watershed is thereby visually identified as "nearest." The mark also seeds a high priority reverse flood (light brown) to overwrite and terminate the search-flood, so it will not go on and mark other watersheds.

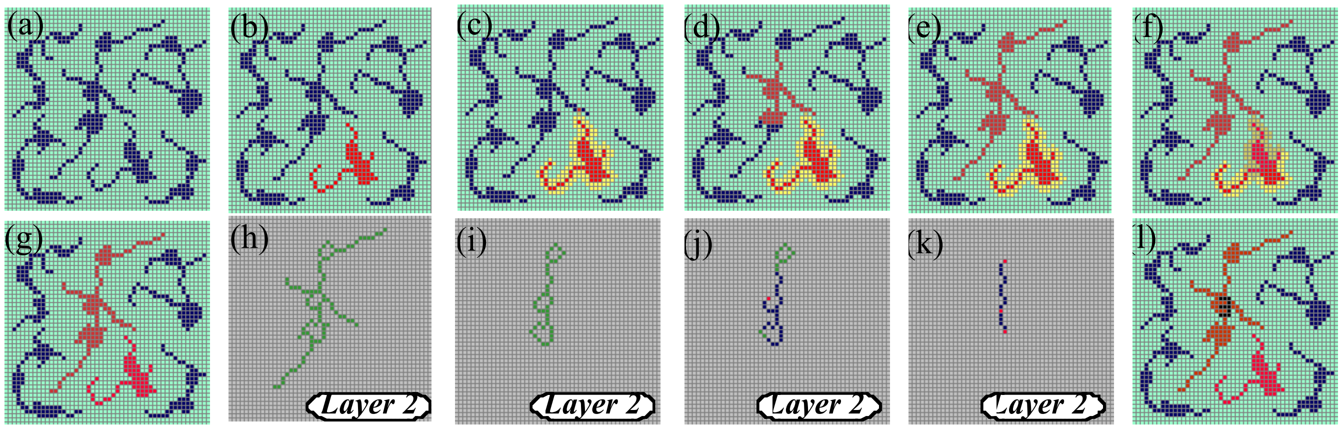

Figure 3. User analysis of exotic species invasion threat. (a) The "Lake District." (b) Infected watershed. (c) Pixel "search"-flood has expanded to find nearest neighbor. (d,e) Nearest, once touched is re-colored. (f) Reverse-flood stops the search, with result (g). To count nearest's lakes, (h) they are copied to a second layer, and hollowed. (i) Branches are nibbled, (j) cycles identified with a blue flood, and cut, each leaving one marked tip. (k) Marked tips are contracted and, when finally clustered locally, are easily counted. (l) The result, "3", is displayed on the top layer for the user. (See Color Plates.)

Suppose she now wishes to know how many lakes are in the threatened system (perhaps lakes are a special breeding ground for the snail). The pixel level algorithm she launches to do the count is complex, and shown only briefly in Figure 3(h-k). Basically it uses a second, scratch pixel layer below the first where it hollows the lakes to become loops and nibbles away the rivers. Each loop is then broken, with one of its broken ends marked. The marks are contracted together for local counting, and the result displayed to the user back on the original surface. Operations like finding the closest among complex and inter-digitating pixel-shapes, and analyzing certain topological features of those shapes, require computation to work very closely with the pixel representations. Pixel rewrite systems are well suited for providing this kind of end-user functionality.

This "setup, launch and wait" style of interaction is a straightforward way to bring the new shape algorithms to the user – basically at the "unit task level". A more "live" feel to the interaction is possible when the underlying pixel computation is fast and simple, and can keep up with the user's continuous actions. A basic example is shown in Figure 4. If two of the simple rules from Figure 1 are active during the drawing process, the user automatically draws, in real time, purely-8-connected, haloed curves - a capability we will use later. The active rules essentially force a kind of constrained drawing.

In a more complex example, we can allows users to manipulate a shape on the screen as if it were a blob of clay. To do this we couple user interaction to an area preserving deformation algorithm [5] allowing a user to massage a blob of pixels into a desired shape without altering its total area, creating a kind of interactive "pixel-clay" (Figure 5). The user interacts by either clicking near the blob, generating a little "explosion" to deform the "clay", or by sweeping a whole pixel curve creating a wave-front to deform the "clay."

Several aspects of this particular application are noteworthy. First, area preservation (or its 3D version using "voxel" rewrites instead of pixel rewrites – a capability the system can support) is a useful capability. Designers are not infrequently given constraints on total area, volume or weight. A landscape architect may be given a budget for a $10m^2$ garden, and would like to be able to manipulate its shape to some pleasing, even complexly "organic" looking one, knowing the area will always meet the requirement. Agricultural land-use planners of the future may want to transform a hilly terrain in southern China to terrain-following, "organically" sculpted rice terraces. In this 3D shape deformation task, the planners cannot come up with a new landscape that will require bringing in or removing billions of tons of earth; their planned deformations must be volume preserving. Second, such area- (or volume-) preserving properties are difficult to implement in algebraic representations of shapes. Most algebraic representations focus on the position of features of the bounding curve (or surface). Other properties, like the enclosed area are difficult to compute, and hence to preserve under deformation. In pixel rewrites, one must simply ensure that individual rules neither create nor destroy clay-pixels, and start-to-finish area preservation is guaranteed. This illustrates one of the basic points of the underlying research program – that these new sorts of computation

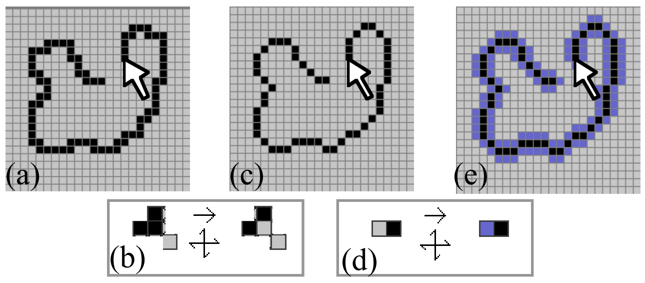

Figure 4. Live-drawing. (a) Simple draw - dropping pixels with no rules active. (b) Making the 8-Conn rule active results in (c) 8-connected real-time drawing. Adding (d) halo-rule, an all-rotations version of the EdgeMark rule from Figure 1(c), yields (e) 8-connected, haloed real-time drawing.

make different aspects of problems available to algorithms, and to human interaction. Plus, there is the added benefit that they all work with any shapes that can be expressed in bitmaps.

USER INTERACTION 2 – NEW WAYS TO CONTROL FUNCTIONALITY

A core goal of our research is to support shape-rich activity using the computational power of rewriting bitmaps - we are always seeking "to bring shape into the picture" metaphorically and literally. This has led, not just to new end-user application functionality, but to new spatially rich ways to control that functionality appropriately.

Such capabilities are often discovered using a "convert and vary" strategy. We take some conventional capability and convert it – re-implementing it using only pixel-rewrites. The mechanisms for the new, pixel version often have a completely different set of natural variants and extensions, compared to the standard version, often ones that indeed "bring shape into the picture".

Consider the conventional capability provided in a GUI interface by a radio button widget. It allows a user to switch between two different application functionalities, e.g., the user clicks Button-1 for pencil-draw or Button-2 for spray-paint. A similar need, of course, arises in pixel rewrite interactions. For example, a graphic designer using the "radar"-based auto-positioning algorithms might want to switch between the auto-converge and auto-disperse functionality.

Following the "convert and vary" strategy, we begin by creating a pixel-rewrite version of the basic "radio button" functionality. Because the later "vary" part of the strategy depends on the internals of the pixel implementation, we go through it in some detail.

First we need a basic control capability, a way to have only one subset of rules firing at a time (e.g., only the "auto-converge" rules and not the "auto-disperse" ones) even though both are nominally active in the system. Whether an active rule fires or not depends on whether its pattern matches somewhere in the field. Thus to effectively turn

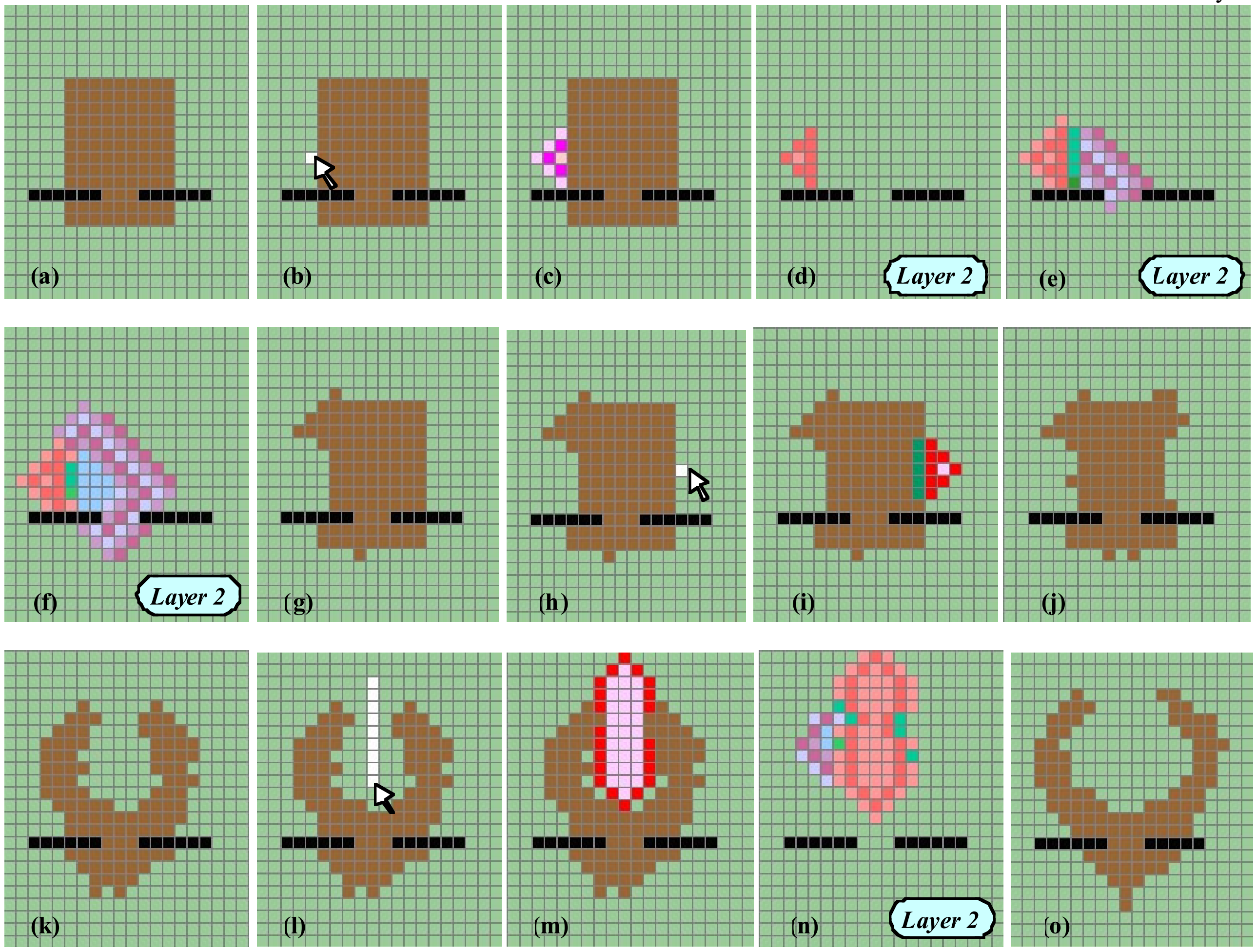

Figure 5. Pixel Clay. (a) Brown clay, black obstacles. (b) Mouse-click. (c) Explosion touches one pixel. (d) Copied to Layer 2, (e) Distance field grows beneath clay. (f) Empty spot found. (g) Clay pixel moved. (h) New click. (i) Explosion, and (j) pixels moved. (k) After more clicks, (l) user sweeps vertical curve. (m) More complexly shaped explosion. (n) Relocation work done in Layer 2. (o) Result, which has same area as original. (See [6] for details of the underlying area-preserving deformation algorithm.) (See Color Plates.)

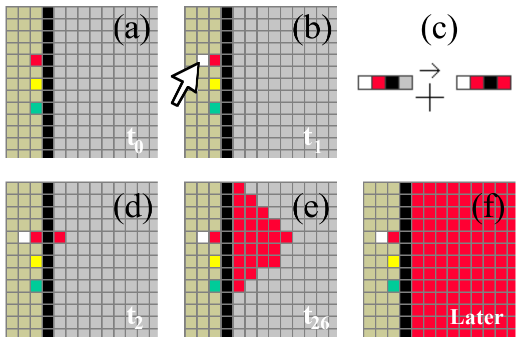

Figure 6. Part of a primitive widget to flood the control region. (a) Initial state. (b) User clicks white by the red "button". Rule (c) then seeds (d) a red flood in (e) and (f), filling the control region. (The "times" in the lower right corners are measured in rule execution counts. I.e., t_{26} means after the 26th rule firing.) (See Color Plates.)

off an active rule we must somehow make sure its pattern never occurs, and do so without tampering with the pixel field of the application. To resolve this seeming paradox, we create a "control" layer, a second pixel layer beneath the application layer. The patterns in all rules are augmented to require a specific color match in the local pixels of this control layer (e.g., red in the control layer pixels beneath all the "auto-disperse" rules' patterns, and green beneath those of the "auto-converge" rules). If the control layer is of the wrong color for a given rule, the rule cannot fire. In this way, subsets can be turned on and off simply by flooding the control layer with the appropriate color.

Providing *interactive* control of which suite of rules is active is then just a matter of writing pixel rules for selectively coloring the control layer in response to user mouse-clicks. We simply set up pixel "buttons", where each, when "selected," colors the control layer correspondingly. Details are shown in Figure 6. The initial state in (a), at time t_0, shows the "widget" to the left of the black bar. The "controlled area," normally a whole separate layer, is here shown as the region to the right. In this case, the three colored pixels are the "buttons". In (b), at time t_1, the user clicks white by the desired color "button". Rule (c) recognizes this pattern and, at time t_2, places a color seed (d) in the controlled area. A simple flood rule operating in (e) and (f) (at times t_{26}, and "later") then fills the controlled area with the desired color. If this were a true control layer, the overlying application functionality layer would then start running its "red" rules.

Finally, to create a real pixel radio-button widget, we need some control logic to provide the XOR functionality, turning off the old button, before turning on the new one. Again because the pixel rewrites are local, the mechanism must work within the basic spatial structure of the pixel array. The implementation (Figure 7) basically involves a vertical-only "flood" that "searches" up and down in space for the old selection. When it finds it, it turns it off and erases its color before flooding the control region with the newly selected color.

Putting all these parts together yields a fully functional radio button capability, where different rulesets, coded by

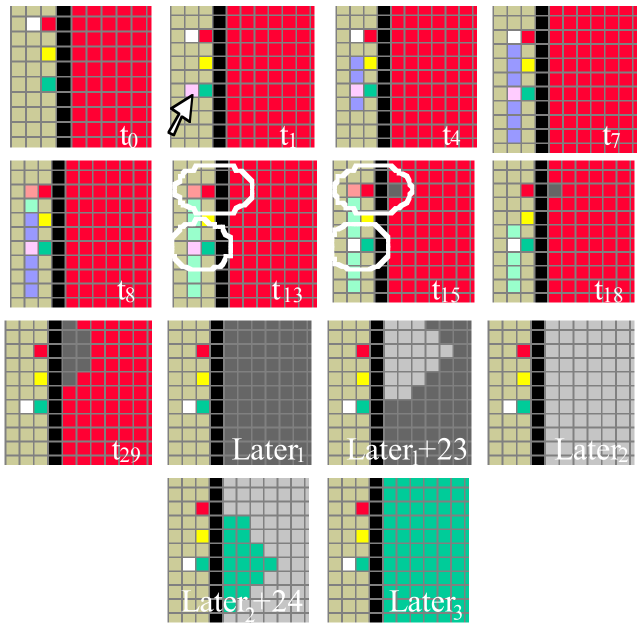

Figure 7. Radio Button Widget implemented in pixel rewrites. Time (t_0) shows the initial state. (t_1) User clicks pink to provisionally assert a new selection. (t_4)XOR vertical SearchFlood goes up & down. (t_7) Finds old selection. (t_8) Marks it and begins FloodBack. (t_{13}) FloodBack completed, stopping search. (t_{15}) EraseFlood seeded and newclick confirmed, turned white. (t_{18}) XOR flood cleaned up. (t_{29}) EraseFlood continues, then ($Later_1$) complete. ($Later_1$+23) NeutralFlood in progress, and ($Later_2$) complete. (The NeutralFlood is needed for technical reasons, to avoid race conditions.) ($Later_2$+24) Finally the new ColorFlood begins and then ($Later_3$) is complete. (See Color Plates)

red, by green or by yellow in their control layer, can be selected by the user's mouse-click next to the corresponding pixel-button.

Although perhaps interesting in its own right, what we have really gained by converting this standard functionality to the pixel domain is the ability to make totally new variations. Specifically, the spatial nature of the implementation mechanism allows natural extensions that "bring shape (represented with bitmaps) into the picture."

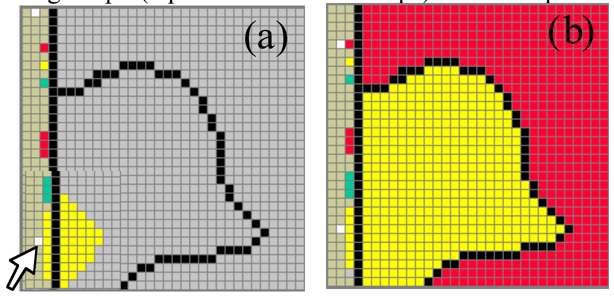

Figure 8. Functionality Maps. "Buttons" can be made beside other parts of the control area, to change the color in different regions. (a) Yellow flood begins for one control area. (b) Two areas, colored for different functionality. (See Color Plates.)

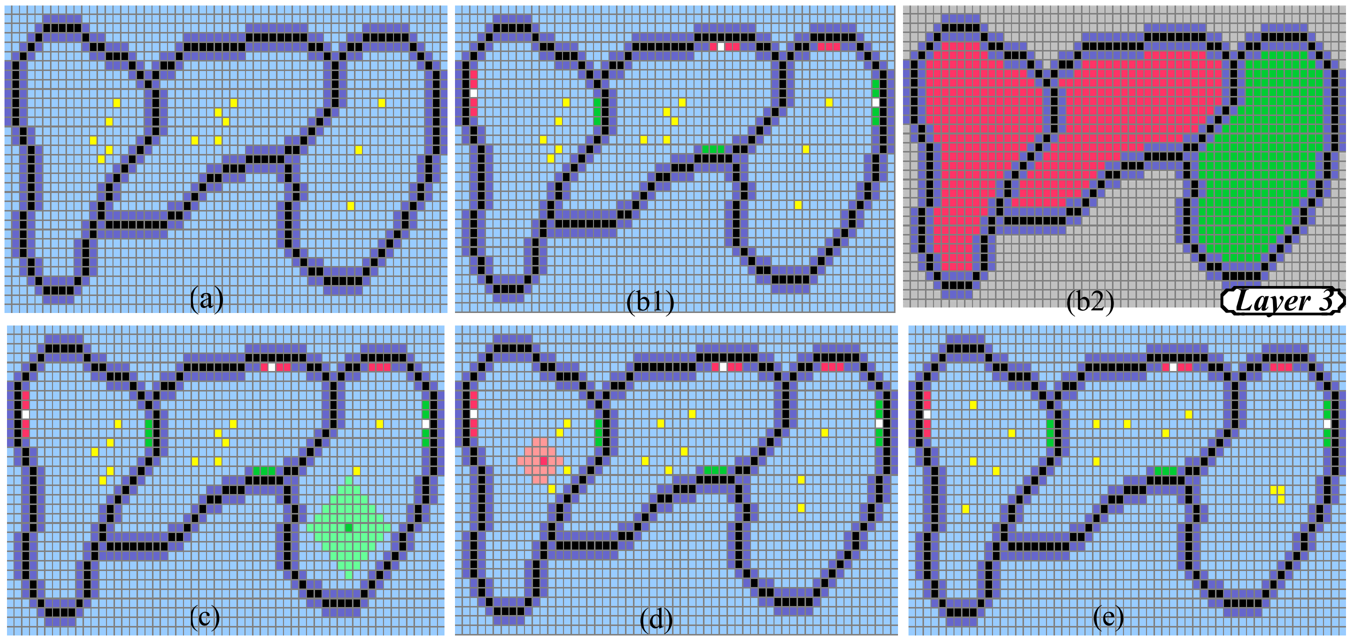

Figure 9. "Organic" radio button widget. (a) The user partitions field of yellow dots by arbitrary boundary curves, then (b1) places red and green pixels in the halos along curve boundaries to serve as "radio buttons", and selects some of them. A second layer (not shown) runs the XOR mechanisms. (b2) Layer3 shows the resulting, correspondingly colored functionality maps. (c) Over the green region a greenish radar-converge pulse spreads out, looking for other yellow dots or obstacles. (d) Over a red region, a reddish radar-disperse pulse spreads out. (e) After time, dots are clumped and spread out respectively. (See Color Plates.)

It is, for example, trivial now for application functionality to have spatial scope. To get the conventional basic control of rule subsets we had to make the appropriate control color available beneath every locale of the application field. Having done so, however, we are now free to make locales differ. If different regions of the control layer have different color, they enable correspondingly different functionality over each. The control layer becomes what we call a *functionality map*. As shown in Figure 8, the radio button mechanism can easily control these arbitrarily shaped regions. One need only draw boundaries to block the propagation of the flooding functionality colors, and make sure the regions have their own radio buttons -- trivially possible by putting sections of the control colors (they need not be single dot-sized "buttons") along the border of the widget region adjacent to the region to be controlled.

These controllable functionality maps, for example, would allow a military General working with a digital map of a mountainous region to indicate that troops in some Regions 1 and 2 should be dispersing as indicated (e.g., as computed by the radar disperse algorithm) while those in Region 3 should be gathering together (e.g., as computed by the radar converge algorithm). This is a kind of upgraded version of bitmasks used in, for example, Photoshop®, to control where some image processing operator will work. Here several regions can be "live" at a time, their boundaries changing interactively by the user in real time, or even computed by other pixel processes working directly with the control layer. And of course, the regions control the arbitrary pixel computations of the application, instead of just photoshop operations. By converting to a pixel implementation, we can vary it easily to provide spatially scoped functionality, a capability quite alien to the conventional widget.

A second variation comes from the fact that the control-logic was implemented in a spatial structure, so that the controllers themselves can be richer in shape, more organically incorporated into the regions they control. Standard rectangular radio-button widgets would typically either obscure the application canvas, or be set off to the side possibly confusing their correspondence to the regions controlled. Here the commanding General working over his map (or the ecologist analyzing her lake district, or an artist working on a canvas) could have the kind of spatially scoped control of functionality described above, but with convenient spatially-local controls – built "organically" right into their arbitrarily shaped boundaries. In Figure 9, the General draws boundaries, perhaps following landscape features (not shown) around various regions containing yellow dots representing troops. Active rules from Figure 4, keep the curves cleanly 8-connected and surrounded by a purple halo, while other simple rules copy those curves down to control layers below. The halo, with its shape following the arbitrary boundaries, serves as the substrate for the radio button internal mechanisms. The XOR "search" flood propagates along the curves halos in Layer 2 (not shown), instead of propagating vertically as in Figure 7. The General simply picks up control colors from a palette at the bottom of the field and places samples of them as radio buttons in any convenient place in the halo of the border of any region to be controlled. Placing a selection dot in any of those "buttons" triggers the underlying region coloring, and turns on the corresponding functionality in that region. In the figure, green starts the radar-converge computation ("troops come together!"), and red starts the radar-disperse computation ("troops spread out!").

DISCUSSION

In this paper, pixel level computations have enriched users' ability to work with any arbitrary shapes in several ways. First, new application functionalities can work with bitmap-shapes, including smoothing curves, pixel clay, and radar auto-positioned shapes. Second, *functionality maps*, dictated by arbitrary bitmaps, allow different regions to behave differently. Finally, *organic widgets,* implemented in the pixels, can be integrated closely with the spatial structure of the user's activity. The almost "raw" spatial nature of the computation increases the spatial richness of the user's interactions.

Future work will in part focus on the underlying shape computation: exploring multi-resolution, non-rectangular and 3D grids, and accumulating more shape manipulation and analysis algorithms. As the work here shows, changes in the computational capabilities lead to changes in opportunities for users' interaction. Another important direction involves exploring how users might best bring this new power to bear. Will there be commonly desired nuggets of pre-programmed pixel functionality, to populate the pixel-rewrite tool palettes of the future? Or will end-user customization and programming be required, with its consequent complexities? Yet another direction we wish to explore is "shape-based" input. The simple mouse-clicks used as input here, seeding a cascade of rewrites from a single point do not have much "shape". One could go much further using video input. Users could use the shapes of their hands to interact directly with the pixel rewrites to further the efforts to bring shape into the picture - interactively.

ACKNOWLEDGEMENTS

This work was supported by grants from NSF (IIS-9877170) and the Intel Corporation. The authors would also like to thank Maria Slowiaczek for her many insightful comments on early drafts, Greg Peters for his programming efforts, Sanjeev Shrivastava and Nikhil Sharma for their varied assistance, and the helpful comments of the reviewers.

REFERENCES

1. Bell, B. & Lewis, C. (1993) ChemTrains: A Language for Creating Behaving Pictures. In *1993 IEEE Workshop on Visual Languages,* 1993, 188-195.
2. Codd, E. F., *Cellular automata*, New York: Academic Press, 1968.
3. Davis, R. & King, J. An overview of production systems. *Rep. STAN-CS-75-524*, Computer Science Dept., Stanford Univ., Stanford, CA, 1975.
4. Furnas, G.W. (1991) New Graphical Reasoning Models for Understanding Graphical Interfaces. *Proc. of CHI '91 Conf. on Hum. Factors in Comp. Sys. 1991*, 71-78.
5. Furnas, George W. and Qu, Yan (2002) Shape Manipulation using Pixel Rewrites. At Workshop on Visual Computation 2002, in *Proceedings of the Distributed Multimedia Systems 2002*, San Francisco, CA, Sept 26-29, 630-639.
6. Furnas, G.W., Qu, Y., Shrivastava, S., and Peters, G, (2000) The use of intermediate graphical constructions in problem solving with dynamic, pixel-level diagrams. In M.Anderson, P.Cheng, V.Harslev (Eds.) *Theory and Application of Diagrams*, Lecture Notes in A. I. #1889, Springer Verlag.
7. Furnas, G., Qu, Y., Shrivastava, S., and Peters, G. (2001) Richer Graphical Interaction using Interactive Pixel Rewrite Systems. Extended Abstract (demo) In *Proc. of CHI 2001 Conf. on Hum. Factors in Comp. Sys.,* 9-10.
8. Kurlander, D & Bier, Eric A. (1988) Graphical search and replace. *Computer Graphics*, 22(4), 113-120.
9. Lindenmayer, A. (1968) Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology,* 18, 280-315.
10. Repenning, A. & Fahlen, L.E. (1993) Agentsheets: A Tool for Building Domain-Oriented Visual Programming Environments. *Proc. of ACM INTERCHI'93 Conf. on Human Factors in Comp. Sys.*, 142-143.
11. Russ, John C. (1998) *The Image Processing Handbook* 3rd Ed, Boca Raton, FL: CRC Press.
12. Serra, Jean (1982) *Image Analysis and Mathematical Morphology*, New York: Academic Press.
13. Siromoney, G., Siromoney, R, Krithivasan, K. (1973) Picture Languages with Array Rewriting Rules. *Information and Control*, 22. Academic Press, 447-470.
14. Smith, D.C., Cypher, A. & Spohrer, J. (1994) KidSim: Programming Agents Without a Programming Language. *Comm of the ACM*, 37(7), 54-67
15. Stiny, G, (1980) Introduction to shape and shape grammars. *Environment And Planning B - Planning & Design*, 7 (3): 343-351.
16. Yamamoto, Kakuya (1996). Visulan: A Visual Programming Language for Self-Changing Bitmap. *Proc. of International Conference on Visual Information Systems*, Victoria Univ. of Tech. cooperation with IEEE (Melbourne, Australia), 88-89.