

# Palimpsest: A layered language for exploratory image processing

Alan F. Blackwell

University of Cambridge Computer Laboratory

alan.blackwell@cl.cam.ac.uk

## Abstract

Palimpsest is a novel purely-visual language intended to support exploratory live programming. It demonstrates a new paradigm for the visual representation of constraint programming that may be appropriate to future generations of keyboardless and touchscreen devices. The current application domain is that of creative image manipulation, although the paradigm can support a wider range of computational expression. The combination of constraint semantics expressed via a novel image-layering metaphor provides a new approach to supporting a gradual slope of abstraction from direct manipulation to behaviour specification. Exploratory evaluations with a range of users give an indication of likely audiences, and opportunities for future development and application.

Keywords: visual programming; visual arts; live coding; constraint semantics

Manuscript accepted for publication in Journal of Visual Languages and Computing, (accepted July 2014)

## 1. Introduction

How are visual *programming languages* related to the software used to manipulate visual *images*? Visual artists often use applications such as Photoshop to create or modify images. This can involve a long sequence of manipulations and transformations. From a programming language perspective, we might think of that sequence as a program: a composition of individual operations whose output is a new artwork. However this kind of visual interaction sequence is more often described as direct manipulation, rather than programming, despite the fact that Shneiderman originally defined direct manipulation in relation to programming (Shneiderman 1983).

The advantage of direct manipulation is that the user can easily anticipate and evaluate the effect of each action as it is taken, rather than waiting until program execution time (or command execution time, in the more familiar comparison between command line and GUI). The disadvantage of direct manipulation is that reduced abstraction and expressive power makes it laborious to repeat the same operations, or variants on them. It is possible, in powerful tools like Photoshop, to construct a simple kind of program in the form of a macro recording – a sequence of operations that can be replayed. However, it is difficult to parameterise a macro recording, or change its behaviour in response to new data or external events.

The goal of this research is to develop an interaction paradigm that shares properties of both image editors and programming languages, offering increased abstraction power to people who work with images. Although the research is theoretically motivated (by the relationship between direct manipulation and visual abstraction), there are a number of possible applications in the visual arts. One of these is the creation of animations, by replaying sequences of image transformations. Another is a kind of version or configuration control, in which different combinations of image transformations might be derived from each other or compared. A third is the performance practice of live coding, in which digital artworks are created in front of an audience, using tools that minimise the separation between run-time and edit-time. Each of these offers increased connection between the program editor and execution environment, in the manner defined by Tanimoto as “liveness” (2013).

This paper describes the design, implementation and evaluation of Palimpsest, a novel image manipulation environment and live visual programming language that demonstrates these properties. The focus on image manipulation revives the long-standing challenge of the "purely visual" language, of which Smith's *Pygmalion* (Smith 1977), Furnas's *BitPict* (1991), and Citrin's *VIPR* (1994) have been creative earlier examples. Although often mooted in this journal and related venues, purely-visual languages have seldom seemed compelling on machines that do, after all, have keyboards. In the age of tablets and touch interaction, keyboards have suddenly become a real inconvenience and hindrance in routine interaction, so this seems a better time than any to explore text-free notations.

The name Palimpsest is inspired by the fact that on touch devices, abstract notations are often superimposed over directly manipulated content. In media and cultural studies, the word palimpsest has been extended from its original sense of a text written over an erased original, to refer to the layered meanings that result when different cultural or historical readings are superimposed<sup>1</sup> (Dillon 2007). The palimpsest thus offers a metaphor for the integration of computational capabilities into the visual domain.

### 1.1. Application example

As a simple example of how Palimpsest might be applied, consider a typical image-editing operation in which a person's face has been separated from the background of an image, and is placed over a coloured frame. The frame is initially red, but the artist decides that the colour should be related to the skin tone in the face. In Photoshop, this would involve using an eyedropper to select the new default colour, and then filling the frame with that colour. However in Palimpsest, the colour can be treated as a visual variable. The variable initially had a constant value of red, but is now bound to a sampled value. The artist then decides that, rather than skin tone, the frame should match the collar of the shirt the person is wearing. In Photoshop, the default colour would be changed, and another fill carried out. However in Palimpsest, the change can be defined simply by modifying the location of the sampled value. Now imagine that the artist likes both of the moods provided by these two colours, and

---

<sup>1</sup> For example in the work *Cambridge Palimpsest* by artist Issam Kourbaj, which was created to commemorate the 800<sup>th</sup> anniversary of the University, and was one of the inspirations for the title.

decides to create an animation alternating between them. In Photoshop this would not be possible. In Palimpsest, the colour-sample location can be bound to a function that changes over time, turning the static picture into a dynamic one contrasting the effect of the two frames.

As an introductory tutorial, illustrating typical operation of Palimpsest, appendix A shows a sequence of screenshots corresponding to the scenario just described. A complete video sequence of the interaction is included in the supplementary materials published with this paper.

## 1.2. Outline of paper

The remainder of this paper is structured as follows. The programming paradigm itself is first described, including an overview of the novel execution model, data types, edit/debug facilities, and support for alternative flow of control and encapsulation. Throughout the discussion, the user-interface design considerations of Palimpsest are discussed in terms of the Cognitive Dimensions of Notations (Green & Petre 1996)<sup>2</sup>. The current prototype implementation is summarised, followed by preliminary evaluation of this prototype with three different kinds of user population. A discussion of related work addresses previous systems that have combined elements of direct manipulation and abstract behaviour specification, as well as related approaches to user functionality and interaction techniques. The final discussion proposes a model for future work of this kind, and sets out an agenda for future research that will develop the Palimpsest concept.

## 2. The Palimpsest programming paradigm

This section provides a high-level overview of Palimpsest operation. An extended step-by-step tutorial introduction is included in appendix B. The supplementary material published with this paper also includes a video demonstrating the operation of these basic aspects.

---

<sup>2</sup> It is assumed that readers of this journal will be familiar with the Cognitive Dimensions framework – in order not to break the flow of discussion, citations are not included on every occasion a dimension is mentioned. However, for quick reference, appendix C provides capsule definitions derived from a popular textbook presentation.

Palimpsest allows users to combine source material (photographs, simple shapes or ink) and treatments of that material (e.g. filters or geometric transformations). Source material and treatments are partially transparent, and are layered on top of one another such that a final image is built up from elements in many layers, just as in professional image manipulation tools such as Photoshop. The behaviour of individual layers can be modified, as usual in such tools, by adjusting parameter values.

However, unlike conventional systems such as Photoshop, parameter values are also represented as image layers, making them first class values in this novel visual language. Interaction with the system involves creating new layers, superimposing them on layers already created, and adjusting values. The resulting stack of source material, values and treatments provides both a visual palimpsest (in the media studies sense), and a layered historical record of the process by which it was achieved. The resulting image can also be viewed in a simplified exhibition mode, with control information hidden and the layers composited together.

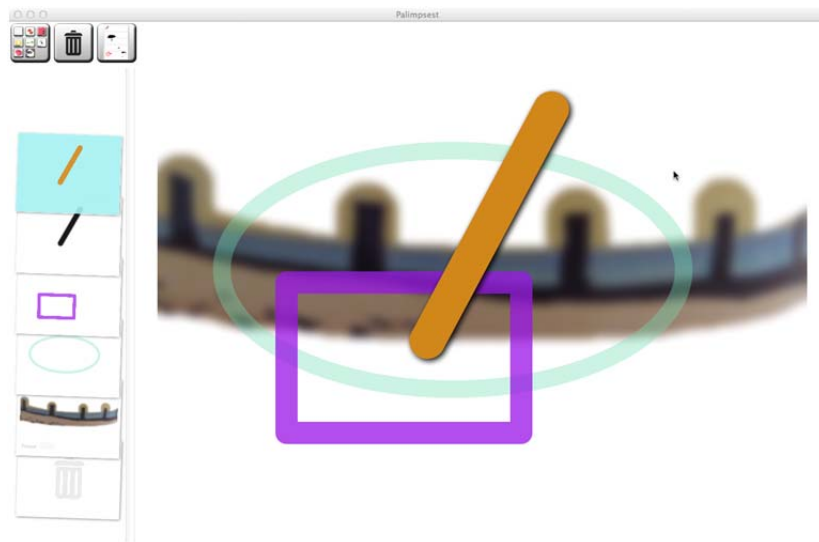


Figure 1 – a screen dump of the main Palimpsest window, with the ordered stack of layers visible as thumbnails at the left. The current layer (a slanted orange line) appears above the other layers in the main view, and the corresponding thumbnail is shaded light blue in the stack at the left to show the location of the current layer within the stack.

The primary Palimpsest display (Fig 1) is an image composed from multiple layers overlaid on top of each other. This stack of overlapping layers is also rendered as thumbnails at the side of the main display, but spread out so that the order of the

individual layers is visible. The current viewpoint can be moved up and down the stack – layers below that point are visible, but not those above (the user viewpoint is of an observer looking down through the stack, with the upper part of the stack behind them). The dynamic visual appearance resulting from this behaviour can be seen in the videos included with the supplementary material.

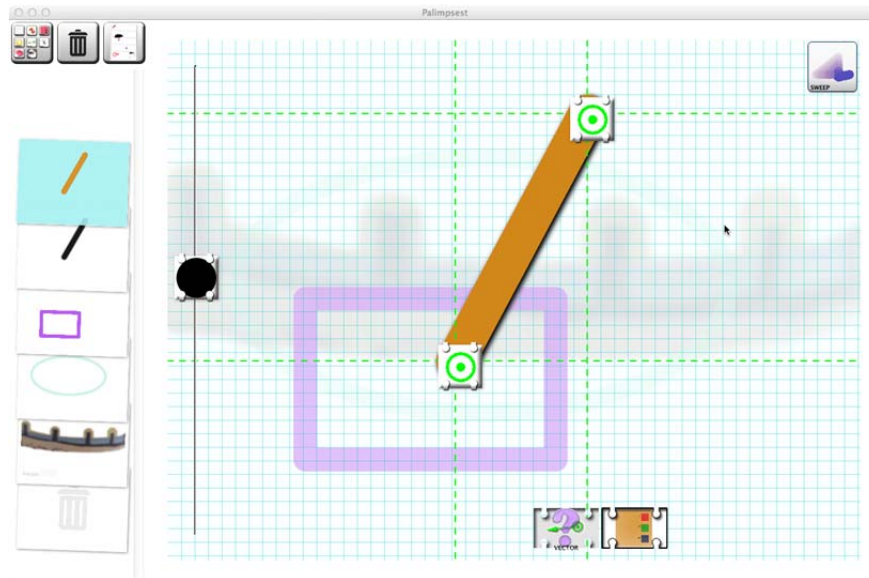


Figure 2 – the current layer, showing typical controls – here a slider at the left, two points in the middle, a button at the top right and two viewports (at the bottom – these are explained later). The visual *content* of this layer is the orange line already seen in Fig. 1, while a magenta rectangle and other content can be seen on the layers below

In addition to its visual content, a layer may also include interactive controls such as sliders, buttons and viewports (Fig. 2). Although all layers have controls, only the controls on the current layer are visible. User interaction with the controls either modifies values or invokes actions.

## 2.1. The stack metaphor

Many actions in Palimpsest result in the creation of a new layer on top of the current one, so that the stack continually grows upward. As a result, the stack has two functions: it presents an intuitive compositional metaphor for building up images from components, and it also provides a history of user actions. By analogy to applications such as Photoshop, the Palimpsest stack could be described as combining the functionality of the Photoshop “layer” palette with the “history” palette.

The convention of providing a navigation panel at the left of the screen, with a larger main panel showing a magnified view, is commonplace in applications such as Powerpoint. However, the Palimpsest behaviour is very different to these, in the way that it represents a stack of superimposed images rather than a simple ordering.

As the number of layers becomes large, the complexity of the image may increase. In order to reduce this visual complexity, layers further down the stack are therefore incrementally faded, making the current working context is relatively clear. As described later, the stack can be simplified by aggregating a group of layers into a collection, with redundant layers removed from display unless specifically requested by the user.

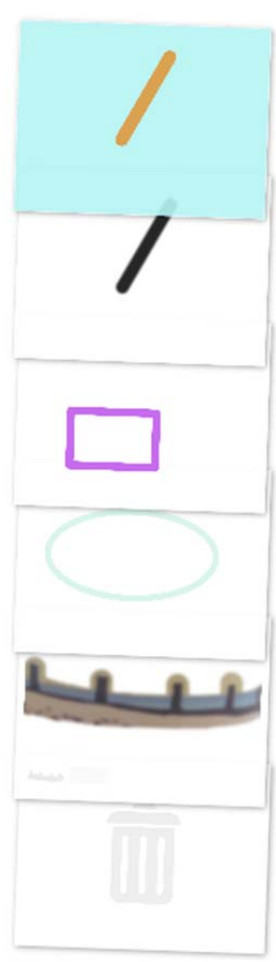


Figure 3 – rendering details of the Palimpsest stack overview. Note reference to the position of the current layer (reflecting the blue grid), and visual depictions of overlap order, tilt, shadow cues, and slight transparency in overlap regions.

The stack overview is rendered as a set of partially overlapping cards, in order to emphasise this difference from conventional applications (see Fig. 3). A further difference with respect to such applications is that, when new layers are added, the stack grows upward rather than downward. This can be understood by analogy to the way that the Photoshop layers palette grows upward, for example after a paste operation (although the same operation causes the history palette to grow downward!).

The choice between these two options (events in time being ordered in either a downward or upward direction) represents a usability trade-off. Earlier versions of Palimpsest did follow the more usual convention, until it became clear that building up layers requires an upward metaphor for the direction of time, just as in geological diagrams. The chosen solution seems currently to be unique in comparison to other systems for visual programming by demonstration. In Kurlander's *CHIMERA* (1990) frames are presented in reading order (left to right), while various others (e.g. Schachman's *Recursive Drawing* (2013)) follow the Powerpoint convention.

## 2.2. Representing constraints between layers

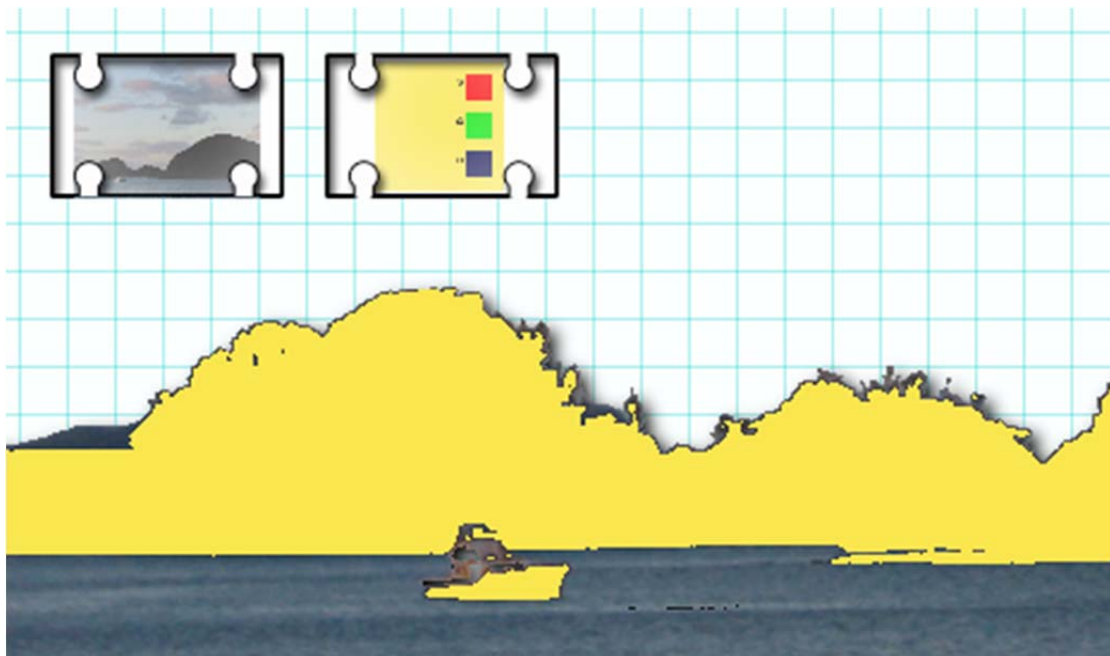


Figure 4 – A thresholding and colour substitution layer, with two viewports showing the layer from which the original image was extracted (left) and the layer defining the substitute colour (right). This example has been created with two operations, each of which defines one of the viewports. The user has started with an original image, and pressed the



“cut out” button. This creates a thresholded image (the same operation is performed at the start of Appendix A, and can be seen in the thumbnail screenshots and supplementary video). The threshold image layer, at the time it is created, contains a viewport showing the linked relationship to the original layer. This is a dynamic constraint – if the original layer changes, the thresholded image will also change dynamically. Second, the user has created a colour layer. When the threshold layer is first created, no fill colour is specified, and the colour viewport is empty. Here, the user has clicked on the empty viewport to create a colour value layer (this operation can also be seen in appendix A, and in the supplementary video). This viewport also represents a dynamic constraint – if the colour value changes, the fill colour in this layer will also change, as seen in the video. Note that this figure has been visually simplified by cropping to exclude a number of elements not relevant to the explanation of viewports, those additional elements include the layer stack, threshold value, and suggestions for exploration, all of which are visible in other figures and explained elsewhere in the paper.

A layer can include one or more *viewports* that refer to a layer elsewhere (Figure 4). The viewport is the primary mechanism for value passing and reference in Palimpsest. It provides functionality that in conventional language paradigms might be implemented as value assignment, value reference or parameter binding. The actual implementation, as discussed later, resembles the constraint pointers proposed by Vander Zanden et al (1994).

In terms of the physical layering metaphor, a viewport can be imagined as a tunnel between layers, allowing the user to look through a tunnel opening in one layer, in order to see another layer at the other end of the tunnel. A scaled image of the referenced layer is shown inside the viewport. To reduce visual clutter, only viewports on the current layer are visible (a trade-off between the Cognitive Dimensions of *visibility* and *hidden dependencies*).

The role of viewports in Palimpsest can also be considered by analogy to cell references in a spreadsheet formula. Each layer in Palimpsest has a single value, just as each cell in a spreadsheet has a single value. A layer is thus analogous to a spreadsheet cell. A spreadsheet cell can contain a formula, including a reference to the value of another cell. A viewport in Palimpsest is analogous to that reference. If the value of that other cell changes, the value of this one will change accordingly. In Palimpsest, if the value at the other end of the viewport changes, the value of the

current layer will change accordingly, as seen in Figure 4. As in a spreadsheet, layer values are constantly recalculated to maintain these constraint relationships.

Viewports can be created in three ways. When a new layer is created, it often represents some transformation of the layer below (for example, blurring or masking an image). The blurred layer includes a viewport that can be used to return to the source image, even if it is hidden or moved elsewhere in the stack. The second way to create a viewport is to drag any layer from the stack onto the current layer. It can be dropped on the background to create a new reference, or dropped onto an existing viewport to replace that reference. The third way is that many layers include possible parameters or default values that are represented as empty viewports. Clicking on an empty viewport creates a new layer of the required type.

### **2.3. Data types to support image manipulation**

The “output” of Palimpsest is the image produced by alpha-compositing<sup>3</sup> all the visible layers in the stack. However those layers may include pixel data (e.g. photographs), geometric shapes (e.g. rectangles or ellipses), text (words or paragraphs) and freehand ink strokes. Furthermore, most layers can be modified by manipulating controls and/or specifying value constraints via viewports.

---

<sup>3</sup> Alpha blending is a standard technique in computer graphics that combines multiple image layers based on the degree of transparency specified for each pixel. This is normally referred to as the alpha channel, and is encoded alongside the RGB channels in the image bitmap.

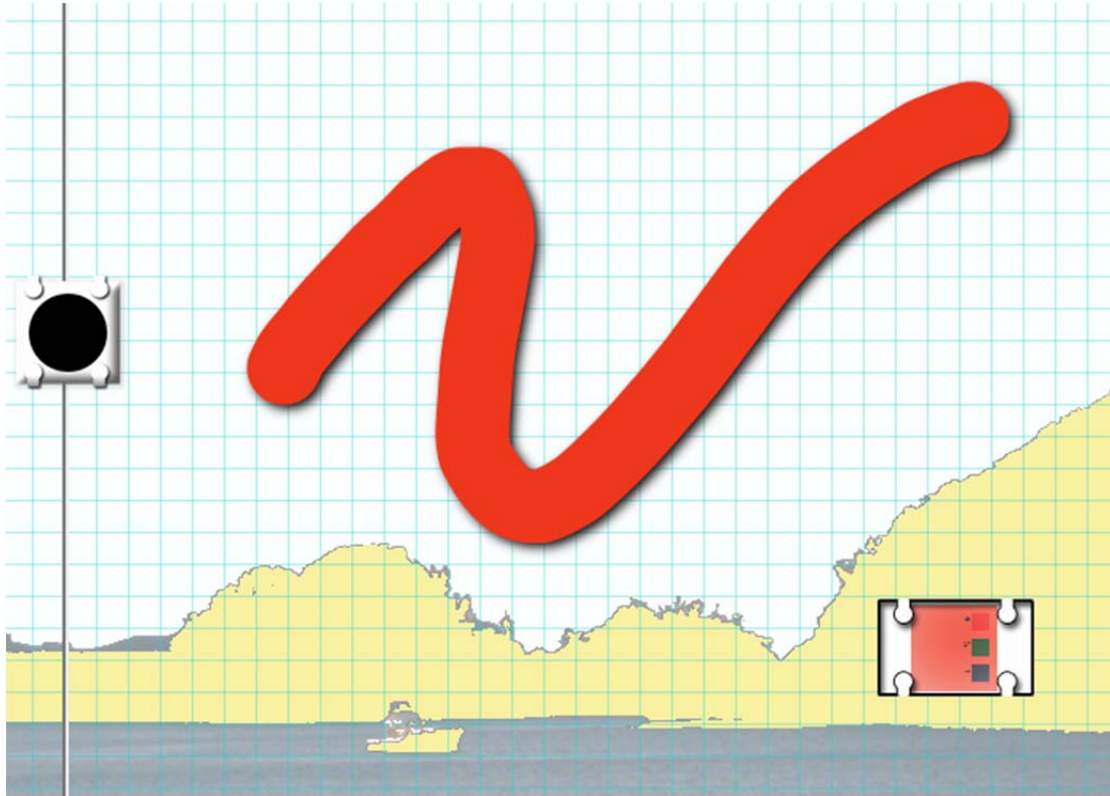


Figure 5 – a freehand ink layer, including a control slider to determine stroke width (left) and a viewport to determine colour (right)

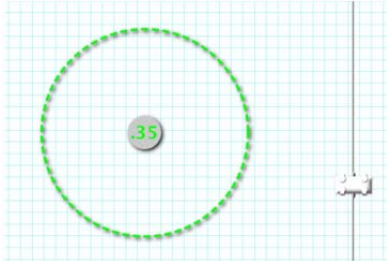
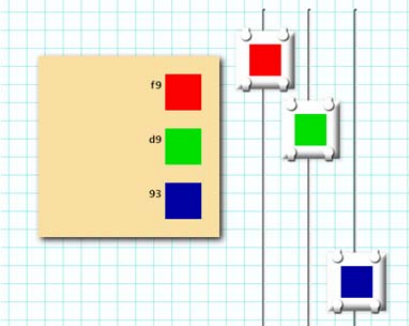
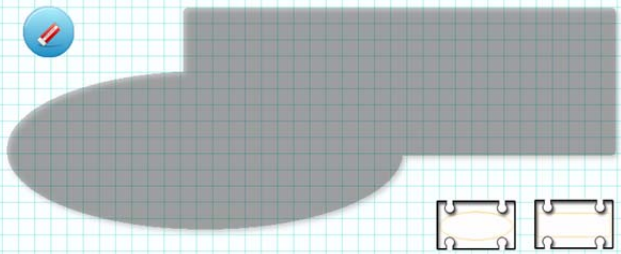
To illustrate a typical effect, the appearance of shapes, text and ink can all be constrained to use different colours or stroke sizes (Fig. 5). Image treatment layers such as blurring, thresholding or fading, and geometric transformations such as rotation, translation and scaling can also be customised with value constraints modifying their parameters. As in any programming language, the viewports specifying these constraints must match the data type of the expected parameter.

The set of types was refined throughout the development of Palimpsest, in order to support sufficiently general geometric and image manipulation functions while also offering an immediate visual interpretation. A key design objective was to escape reliance on mathematical abstractions. For many arts practitioners these seem to be an obstacle to adoption of computational methods – arts practitioners have often found school mathematics challenging, and many avoid situations in which mathematical abstractions or notation are routinely used (Church et al 2012).

The resulting set of visual value types is: *Shape*, *Image*, *Mask*, *Colour*, *Amount* (as a proportion 0-1), *Point*, *Direction*, *Count*, *Rate* and *Event*. Each of these can be

represented by a value layer that provides a visual representation of the value, with direct manipulation controls that allow the user to adjust and explore the appearance of values throughout the range. In addition to manually exploring the value range, the user can also drop a value onto any slider handle, defining a constraint relationship. This provides a transition between direct manipulation and program control.

Examples of value layers and controls, with the design rationale for each, are provided in table 1.

<p><i>Amounts</i> are represented by an expanding circle whose area can cover the whole screen (a value of 1) or shrink to a point (a value of 0). They can be interactively adjusted using a slider, to explore the appearance of values throughout the range.</p>	
<p><i>Colour</i> values are adjusted with three slider controls for each of the RGB values<sup>4</sup>.</p>	
<p><i>Masks</i> can be defined as a union of filled regions, shapes or thresholded images. Viewports maintain links to the original shapes, which may change dynamically. The eraser button at the top left can be used to selectively erase part of the mask, which will override that part of the resulting union.</p>	

<sup>4</sup> Rather than a didactic explanation of RGB colour space, users acquire understanding of the space by exploring the effects of manipulation (each slider handle shows the brightness of that component, and a sample patch shows the blended result). Note that it would have been equally easy to apply the same approach to HSB or CMYK sliders – and that those colour spaces may be more relevant to existing graphic design applications. Palimpsest aims to support creative exploration of computational concepts, so RGB has been retained precisely because it reflects technical and scientific conventions.

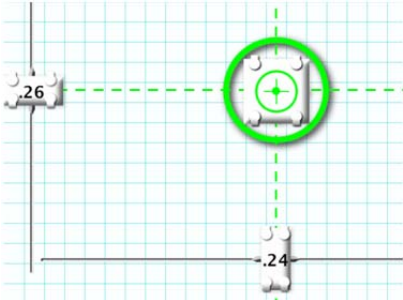
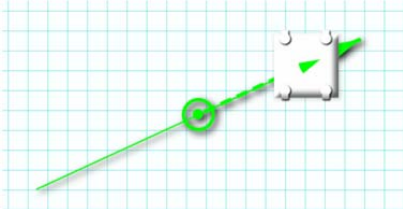

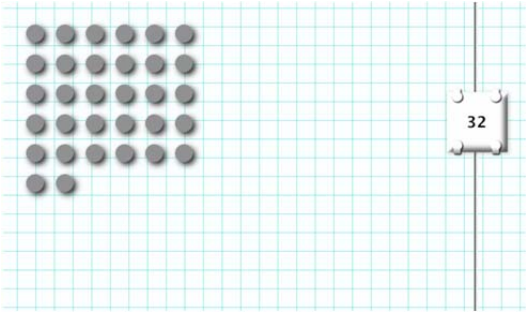
<p>A <i>point</i> is a draggable handle that simply represents its own location anywhere on the screen. X and Y components are available as sliders at the side of the screen, whose relationship to the main control location is made clear by the common fate of their constrained movements.</p>	 <p>The image shows a central point control consisting of a white square with a green crosshair. This point is surrounded by a green circular highlight. To the left of the point is a vertical slider labeled '26', and below the point is a horizontal slider labeled '-.24'. Dashed green lines connect the sliders to the point's X and Y coordinates on a light blue grid.</p>
<p><i>Direction</i> is represented as a compass heading, with a needle whose length varies with magnitude. Mathematically, the result can be used as a vector, although the user need not be aware of this.</p>	 <p>The image shows a compass needle on a light blue grid. The needle is a green line with a circular head at the tail and a white square with a green arrowhead at the tip. The needle is oriented diagonally upwards and to the right.</p>
<p><i>Rate</i> is visualized as a moving sweep hand, controlled with a magnitude slider that changes the speed of the rotating hand</p>	 <p>The image shows a circular dial with a grey sweep hand. To the left of the dial is a vertical slider labeled '1.0'. The dial is on a light blue grid.</p>
<p><i>Count</i>: There is little need for precise counting in the visual domain of Palimpsest. Distinction between 1, 2 or 3 items is significant, but the interval between (say) 892 and 903 is less important. Count values are therefore controlled via a logarithmic slider. Although the precise number can be seen, the main visualization is a square arrangement of dots, indicating magnitude at a glance.</p>	 <p>The image shows a grid of grey dots on a light blue grid. The dots are arranged in a roughly square pattern, with 5 rows and 5 columns, and a few extra dots at the bottom left. To the right of the grid is a vertical slider labeled '32'.</p>

Table 1 – types of value layer supported in Palimpsest, with associated design rationale.

## 2.4. Learning to use constraints by exploration

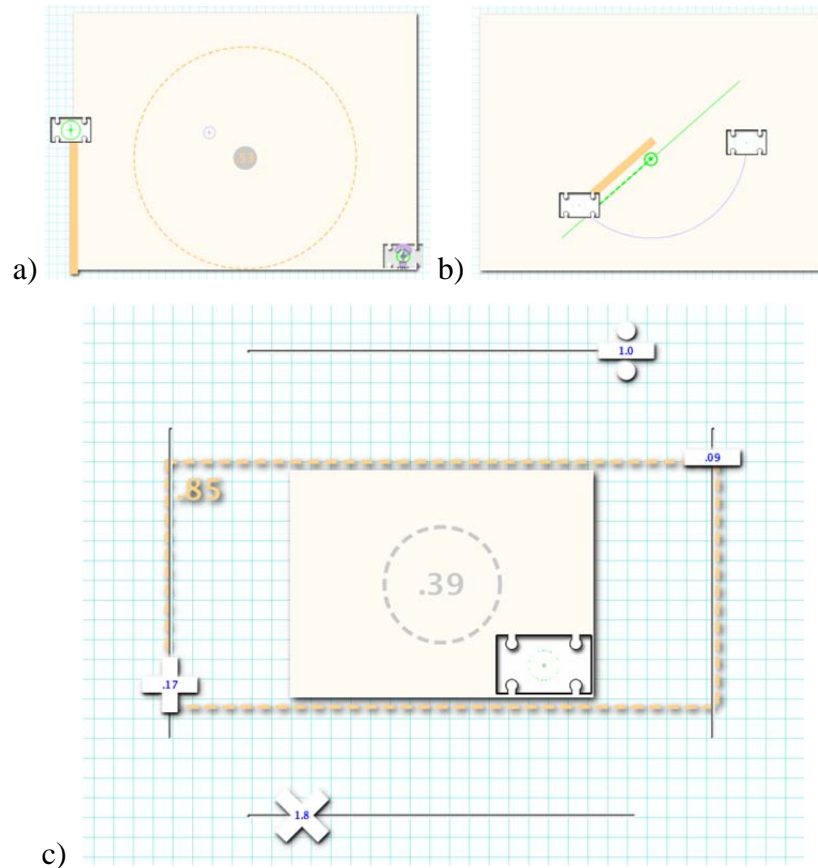


Fig 6 – value type conversions: a) an amount defined by height of a point – the vertical bar at the left shows the height being measured, and the large circle indicates the derived amount; b) a direction vector defined by angle and magnitude – the sloping bar in the centre shows the resulting vector, based on angle and magnitude viewports; c) arithmetic scaling of an amount – the area of the circle in the middle shows the original amount, and the large dotted rectangle indicates the derived amount. Addition and subtraction change the height of this rectangle, while multiplication and division change its width. These controls can be explored and directly manipulated in any order, or bound to values from other layers by converting them to viewports.

Previous experience with developing programming systems for use by artists has found that anxiety about mathematical operations such as trigonometry has been a severe obstacle to simple visual manipulations. In Palimpsest, mathematical operations are therefore presented in the same way as simpler value definitions – the behavior of an operation can be explored by direct manipulation of sliders, and other values can be dropped onto those sliders for programmatic control. Examples are shown in figure 6 of use of coordinates to convert between points and amounts,

simple trigonometry to convert between amounts and directions, and a four-function calculator that visualizes arithmetic operations as the changing area of a rectangle.

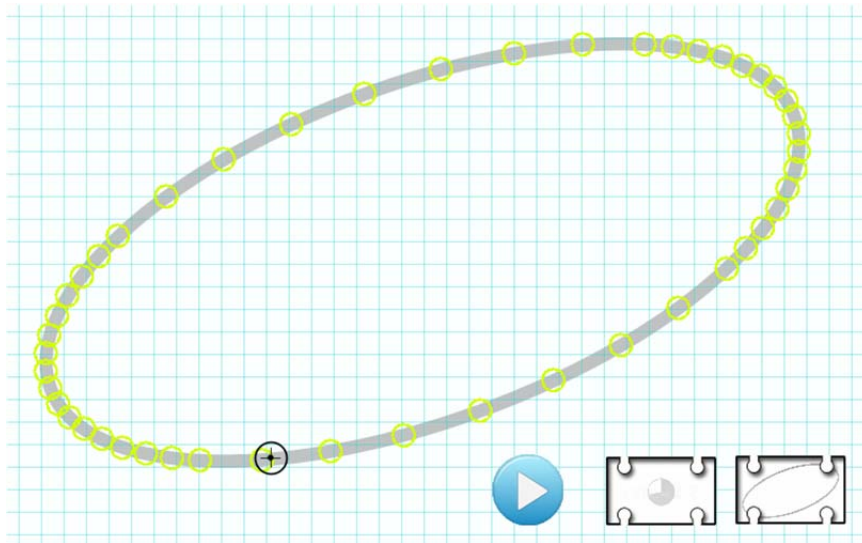


Fig 7 – an animation path specified to follow the shape defined on an ellipse layer (the viewport on the right), at a rate defined on a rate layer (the viewport on the left).

More complex behaviours are also presented as conversions between value types – for example, a motion trajectory can be defined by deriving a moving point value from a shape, ink or mask layer (Fig. 7).

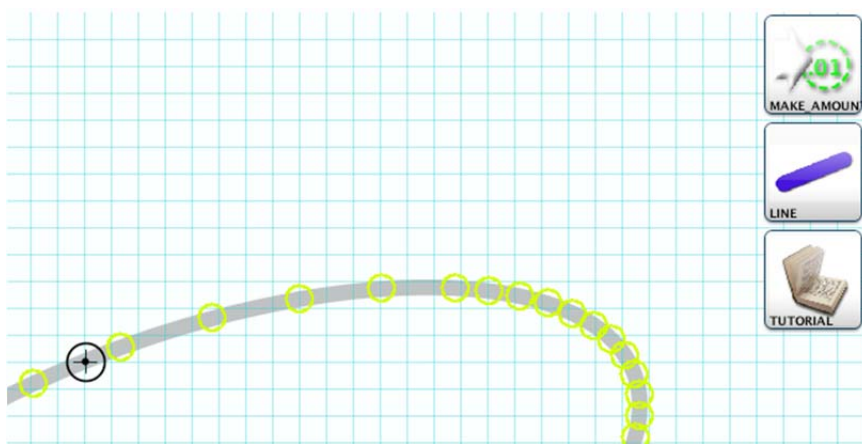


Fig 8 – three buttons suggesting layer types for further exploration (top right). This type of layer provides an animated point, as in Fig 7, which might usefully be used to create an oscillating value based on the height of the point (top button), or as a point defining one end of a line (middle button). Every layer also includes a button that can be used to request a brief tutorial explaining the behaviour and use of that type (bottom button – the tutorial will appear as a new layer).

Exploration of Palimpsest functionality often involves experimenting with these conversion layers. To encourage exploration, each Palimpsest layer includes a small selection of other layer types that it might be interesting to create next. These are offered as a changing set of control buttons at the top right of the window (Fig. 8). Pressing one of these creates a new layer of that type, with a viewport linking it to the layer where it was created. Most layers also provide a link to a two or three sentence tutorial description, naming the layer type and its general capabilities.

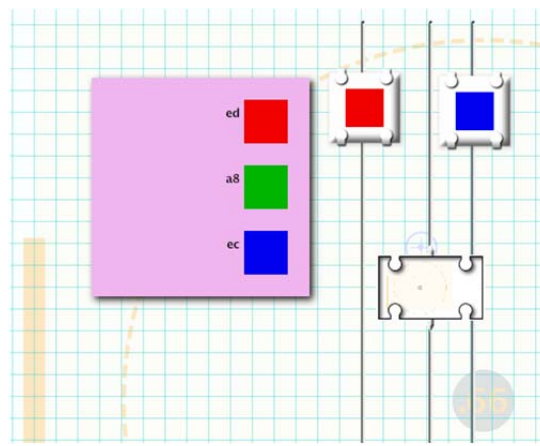


Fig 9 – the green component of a colour value layer has been constrained to follow a derived amount (visible on the layer underneath). The slider for the green component has been replaced by a viewport.

Alternatively, type conversions can be created dynamically in response to the user dragging and dropping a layer of one type onto a control or viewport of another type (Fig. 9). This is often useful as a means of creating values that change dynamically – via a constraint relationship to an animated point.

## 2.5. Deriving abstract values from direct manipulation

Many programmable systems become hostile to exploratory design through a combination of the Cognitive Dimensions of *abstraction hunger*, *premature commitment*, and *viscosity*. An earlier version of Palimpsest inadvertently demonstrated these faults, because geometric transformations such as rotation were specified by first creating a treatment layer to rotate an image, then a viewport to specify the vector. The *abstraction hunger* resulted from the fact that the user needed to request a mathematical operation as the first step. *Premature commitment* resulted



from the fact that a choice had to be made as to which operation was required. And *viscosity* resulted from the fact that to change the operation (e.g. from rotation to scaling), the layer had to be deleted and a new one created.

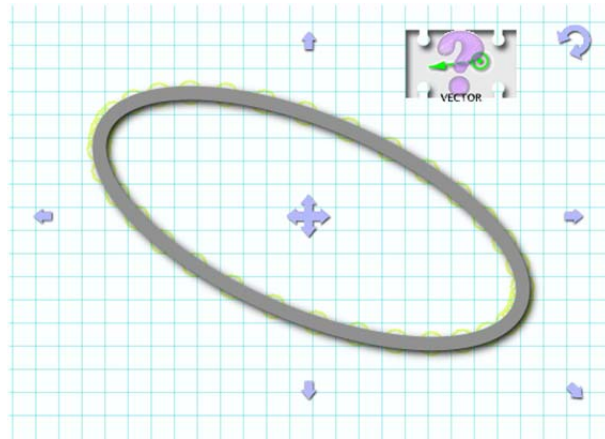


Figure 10 – an ellipse has been rotated using direct manipulation, by dragging the curved arrow handle (top right). This action has resulted in the appearance of an empty viewport for specifying the rotation vector (to the left of the handle). This viewport can then be used under program control to modify the rotation value, as an alternative to further direct manipulation.

The revised approach to geometric transforms illustrates the Palimpsest philosophy of supporting exploration through direct manipulation. All shapes and image layers were given conventional “handles” for direct manipulation of the image via scaling, rotation and translation. In most image editing software, these handles result in affine transforms that are stored internally to the software, but whose parameter values are never explicitly revealed to the user. In Palimpsest, the abstract parameters resulting from the direct manipulation are exposed to the user, so that the same transformation values can also be applied elsewhere.

When the corresponding handle is used, a viewport appears on the layer, representing a link to the transformation that has just been specified by direct manipulation (Fig 10). The user can either ignore these viewports, or can click on them to add a new value layer to the stack. That layer will continue to change its value as the shape is directly manipulated, but it can also be used as an abstract label for programmatic manipulation of the shape, or constraining other layers to have the same value.

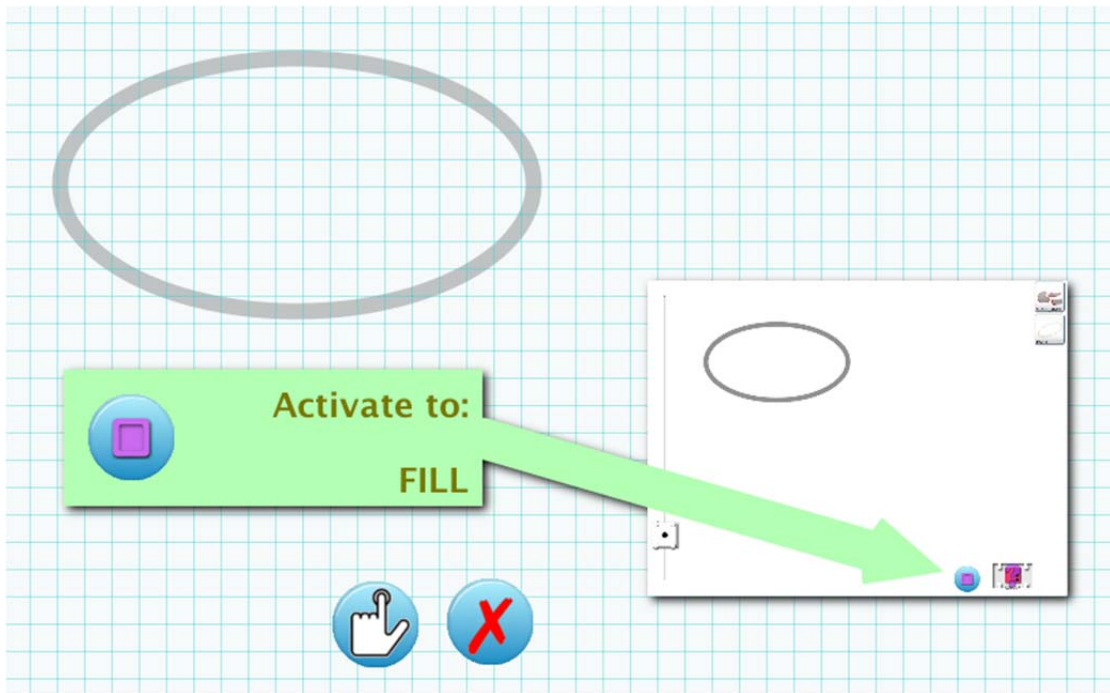


Figure 11 – an event layer corresponding to the button action that would fill the ellipse on the layer below (the fill button is a standard mode control on the ellipse layer). Controls on the event layer either activate the action (bottom centre left) or disable the relationship (bottom centre right).

An emergent overall design principle for Palimpsest, was to ensure that any direct manipulation control can also be used to define a new layer which duplicates the effect of that control (Fig 11). This applies not only to value controls such as sliders and directions, but also to menu buttons, mode controls and operations that change the stack. This is inspired by radically user-customisable systems such as GNU Emacs, in which every user action is bound to a LISP function that could, in principle, be used to automate or customise that action.

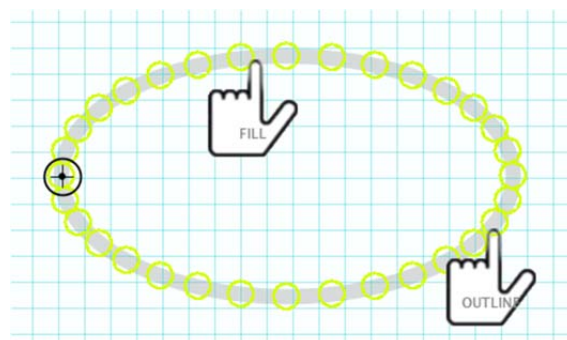


Fig 12 – links to two event layers that will be triggered at defined positions along an animated path.

Simple button clicks are therefore represented by *event* layers showing the button to be clicked. Event values can be triggered manually, synchronized or repeated for a specified count or rate. They can also be dragged onto any animated path, allowing users to specify the ordering of events as a spatial sequence of any shape within a visual context (Fig. 12). Most conventional program control constructs can be emulated using these purely-visual conventions. They would not be particularly useful as a general purpose programming language (the Cognitive Dimensions include high *diffuseness*, poor *juxtaposability*, and high *viscosity*), but they are effective in providing a gentle slope from direct manipulation to more powerful abstraction.

## 2.6. Support for computational abstractions

Much of the image processing functionality of Palimpsest can be achieved using simple Boolean, arithmetic and geometric operations over image and shape values, with a small library of built-in image filters (Fig. 13).

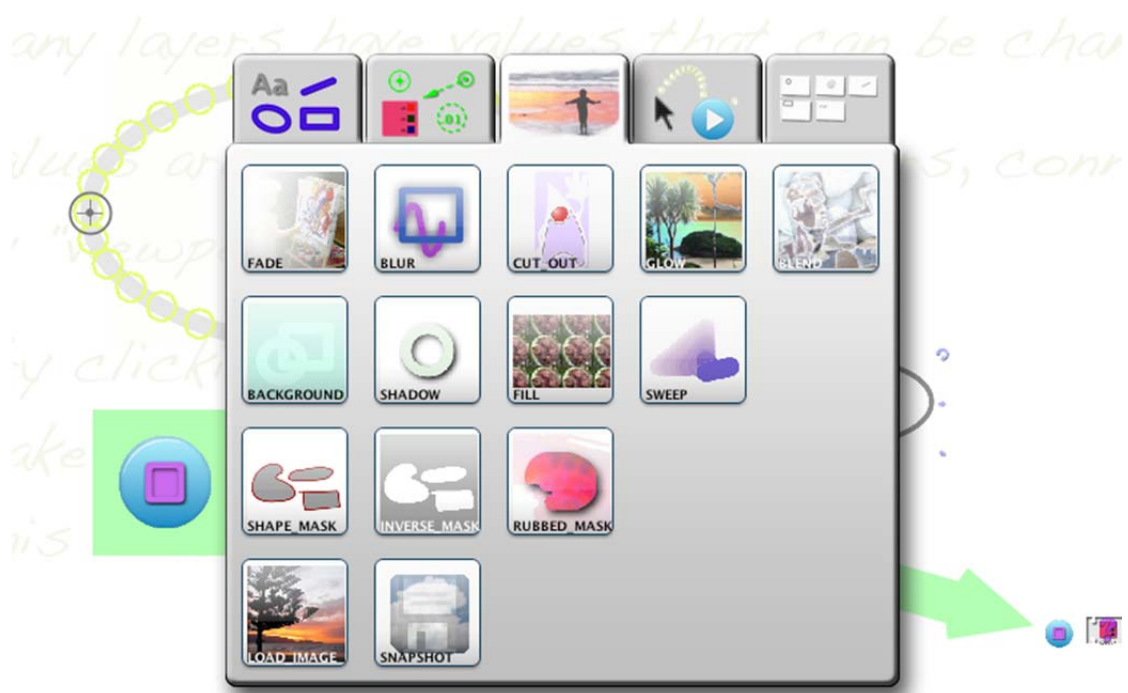


Figure 13 – screenshot of the Palimpsest menu layer, showing one of five tabs that can be used to create a variety of image treatments and layer types

However, a further research objective was to investigate the extent to which more powerful features can be presented within this general strategy of starting with direct

manipulation, and providing a gentle slope of more abstract computation. Experience with deployment of spreadsheets suggests that facilities of this kind might not be explored by all users, who find the basic automation facilities of cells and formulae already provide sufficient benefit over direct manipulation. However, power users, whether self-taught or transferring knowledge from other programming environments, are likely to recognise the potential of general computing functionality.

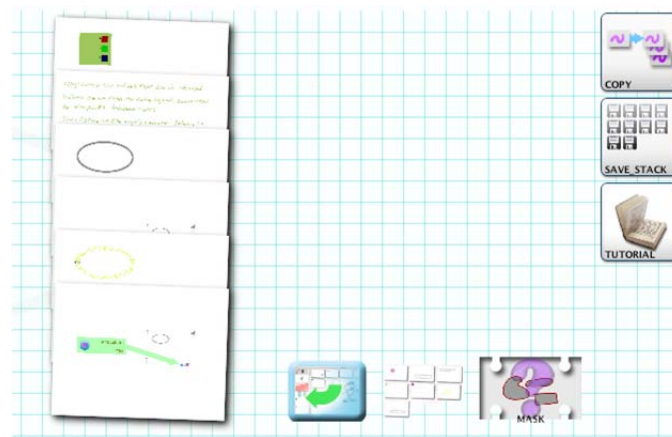


Fig 14 – a section of the Palimpsest stack collapsed into a single layer. The buttons at the bottom load this layer into the main stack (left), spread out the thumbnails into a grid so that individual layers can be selected or masked (middle), and allow a mask to be specified (the empty mask viewport at the right). (Buttons at the top right suggest other types for exploration).

In order to support simple data abstraction, sections of the stack can be collapsed into a single layer, representing a collection of other layers (Fig. 14). This collection is rendered in the same way as the stack overview, with each member shown as a card containing a scaled version of the layer contents. The image value of the collection layer itself, however, is not the arrangement of cards, but the composite result of the layers that it contains. This means that collection layers can be used for encapsulation and abstraction of image-producing functions. They include a persistence mechanism, by which stacks can be saved to disk as Palimpsest “programs”, and an editing interface – the main stack can be exchanged with the contents of a collection, allowing the behaviour of that collection to be explored and modified. Members of the collection can be selectively disabled (either to be “commented out”, or conditionally executed) by drawing a mask over the collection (Fig. 15). This mask can, of course,

be dynamically modified under program control via a viewport referring to a mask layer.

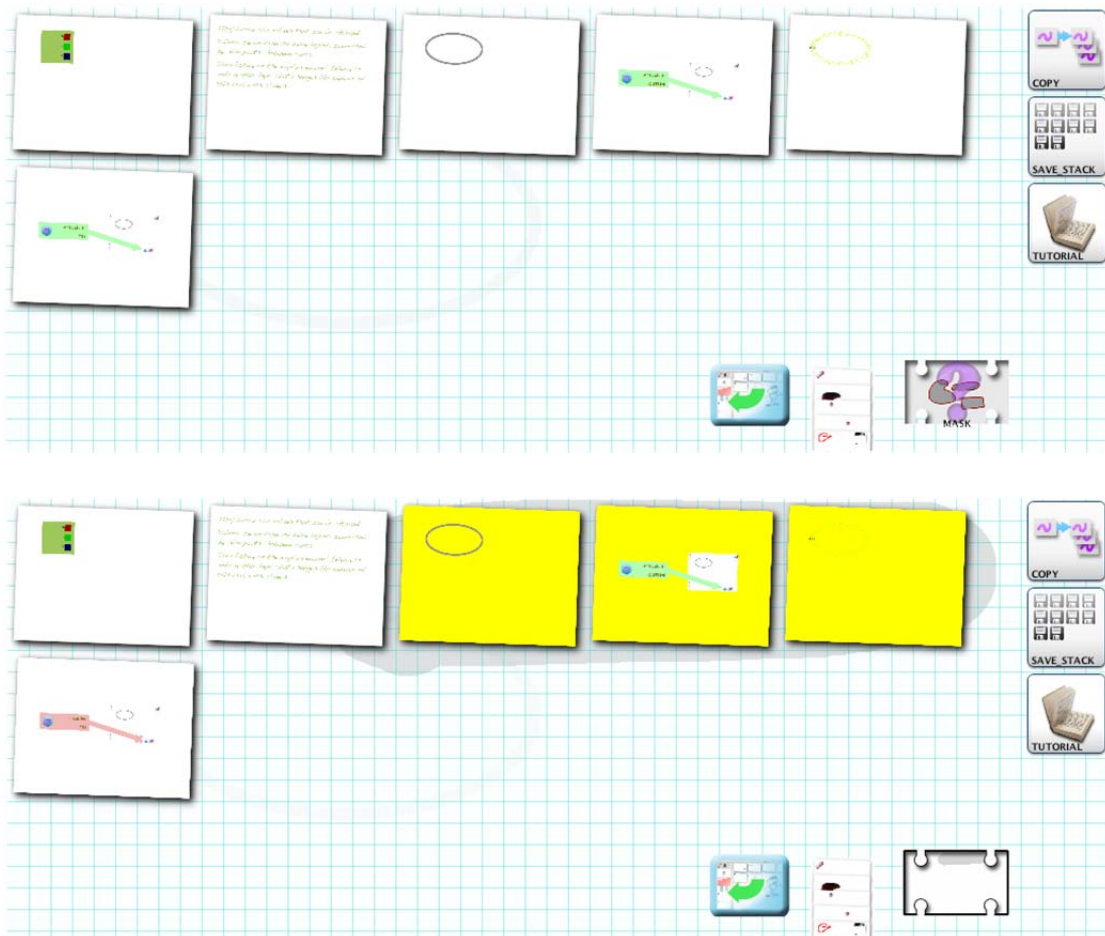


Fig 15 – a) collapsed section of the stack arranged as a collection in a grid of thumbnails, and b) use of a mask to selectively enable and disable layers within that collection. Other control elements appearing in this figure are the same as described in the caption of Fig 14.

Simple conditional expressions allow masks to be enabled and disabled based on a value comparison (Fig. 16). Viewports can also be enabled and disabled under user control (or by an event layer carrying out the same action). Alternative values can be selected, with dynamic typing so that the user can create a layer that connects one of two different value layers to a viewport. All of these facilities allow conventional computation algorithms to be implemented, but as with visual systems such as *ToonTalk* (Kahn 1996), *AgentSheets* (Repenning & Sumner 1995) or *BitPict* (Furnas 1991) in a manner that is rather less convenient than textual symbolic notations.

Typical challenges in such systems include Cognitive Dimensions of *viscosity*, *role expressiveness* and *hidden dependencies*.

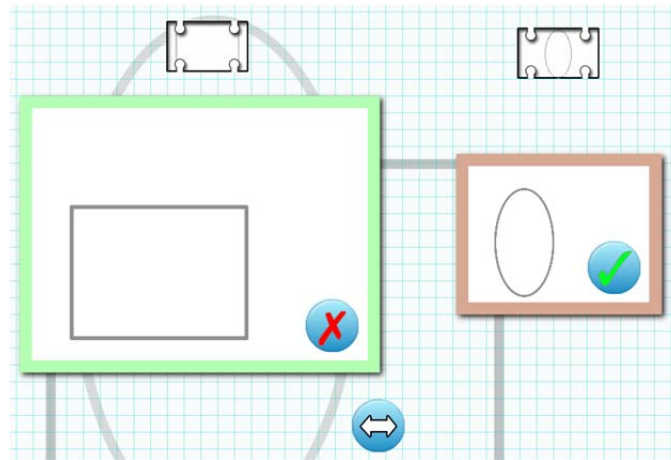


Fig 16 – layer to select a value between two alternatives, each of which is a value obtained from another layer via the viewports at the top. Either of the two alternatives can be chosen by direct manipulation (clicking on the check mark button at the right in this screendump), or the chosen alternative can be toggled (using the toggle button at the bottom). These actions can be made conditional on other values by creating an event layer to actuate the direct manipulation buttons.

The most complex facilities in the current version of Palimpsest are those that create new instances of layers under program control. Instantiation is a conceptual abstraction that has proven to be an obstacle in other end-user programming systems. The standard version of *Scratch* (Resnick et al 2009) for example, does not provide any mechanism to create a new sprite instance under program control. Several attempts have been made to address this problem in Palimpsest, none of which are likely to be acceptable to any but the most determined power user. However, the currently most usable approach (included in the present distribution) is a layer that can make copies of other layers, or of collections. Where the layer being copied includes bindings to other layers via viewports, the user must specify which of those bindings should be inherited in the new instance. At present, this can be done by drawing a mask over the bindings that should be retained.

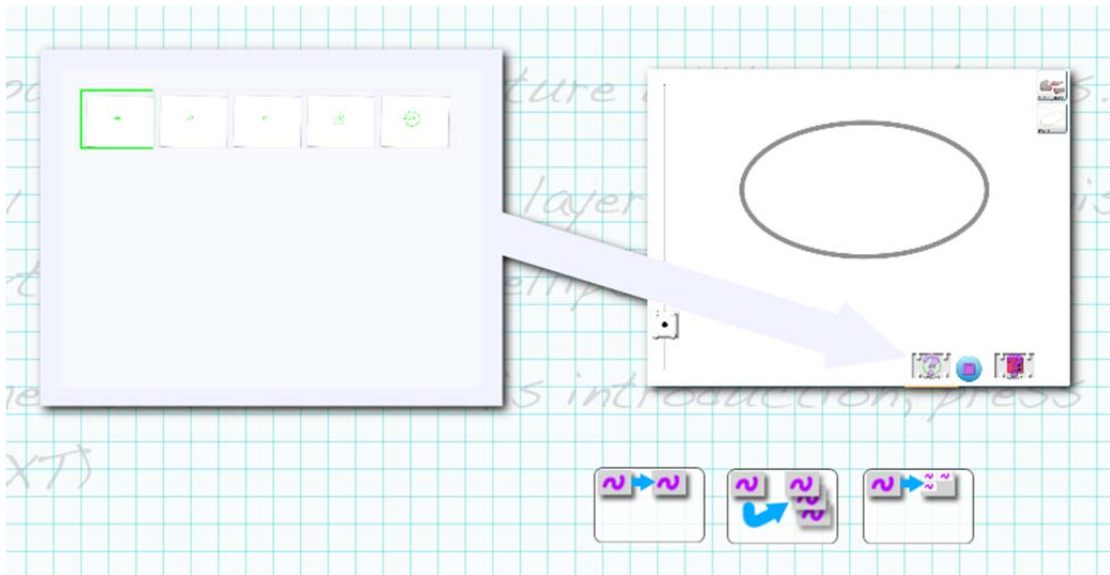


Figure 17 – a control layer to create multiple instances of an ellipse (as specified at the right), each with a different value substituted from within the sequence shown at the left. The buttons at the bottom can be used to request either a single instance (left button, indicating the next element in the series), multiple instances (middle button, indicating one instance for each element in the series), or a collection containing multiple instances (right button).

In principle, the copy mechanism offers one of the major benefits of abstract interaction over direct manipulation – the ability to repeat operations more easily. In practice, the most useful such facility in Palimpsest has been achieved through the ability to drag a collection of layers into a viewport. When this is done, any members of the collection whose type matches the viewport are used to generate a map operation, in which another copy of the target layer is made for each matching member (Fig. 17). As with other copy operations, users have the option to place these copies directly on the stack, or into another collection. In combination with the viewport masking facilities described above, this could provide a simple version of visual functional programming, with curried parameters and map operators. However, as before, this would be sufficiently awkward to use that it must be regarded as a curiosity, or at best a feasibility demonstrator, rather than a practical tool for regular Palimpsest users.

### 3. Implementation overview

The current version of Palimpsest has been implemented in approximately 40,000 lines of Java code over 300 classes. It relies heavily on the Java *Graphics2D* package, especially the *BufferedImage* and *AlphaComposite* classes, which are used throughout to provide the rendering of semi-transparent layers. The system architecture relies on extensive caching and double-buffering to provide immediate response to user manipulation of value controls, with all animation layers and cache updates carried out within *SwingWorker* threads. There may be multiple layers animated at the same time. Animation updates are therefore dispatched from a single thread to ensure a consistent time base – a strategy also used in Smalltalk.

Calculation of shadows is generally the greatest performance bottleneck – the shadows under the current layer are an essential visual cue to system behavior, so have been rendered using a relatively high quality Gaussian blur. Automated background removal is often necessary for imported images, in order to identify regions of an image layer that can be considered transparent. As with shadow calculation, high quality background removal is essential to user perception of the layer metaphor. Noise filtering, adaptive thresholding, and compensation for illumination gradients are all applied (adapted from the approach used by Wellner (1993)). At present, Palimpsest does not process live video input, although this would be a natural extension. If live input was used, performance of these algorithms would become a more significant factor – at present, images are only processed once on capture, with the user given some control over threshold parameters.

There is limited use of Java Swing functionality, except to manage the scrolling of the stack overview. Although much of the interaction (e.g. binding of layers to viewports) involves drag and drop of display elements, the Java drag and drop API was of limited value, with its emphasis on inter-application data exchange rather than visual representation of the drag (which is assumed to be defined by the device vendor rather than application developer).

As with spreadsheet systems, the data-driven execution model relies on type inference to create valid constraints in response to user actions, followed by change propagation to update values at execution time. Type inference in Palimpsest uses Java reflection to determine the value interfaces supported by any layer, control or viewport, with



type conversion layers created automatically in response to those user actions that require them. Collections, constraint indirection layers, and some unbound viewpoints are polymorphic, meaning that the user is reliant on the “traffic light” cues (a green, amber or red highlight) during interaction to anticipate the result of dropping a value into a specific location.

Both the geometric operations, and the layout of user controls on Palimpsest layers, involved a substantial amount of Java code to manipulate simple point, line and region calculations. Swing layout components are too heavyweight for these purposes, but comparison between the developing functions of Palimpsest and the Java *Graphics2D* geometry primitives made it clear that many aspects of geometry processing in Java are both predictable and repetitive (such as identical operations repeated for x and y coordinates, or parameter lists of the form “ $(x+width, y+height)$ ”). A set of lightweight “natural language” utility classes for Java geometry was therefore implemented. Base classes Location, Offset, Size and BoundingBox extended the Java Point, Dimension and Rectangle, but included all possible constructors and conversions combining them as well as natural language expressions such as bottomRight, shiftLeft, centre, addMargin and so on. Although there are a large number of such expressions in English, there is seldom any ambiguity about their meaning. This meant that a development policy of adding further constructors and operators whenever a layout problem could naturally be expressed in a short phrase rapidly resulted in a robust and generic Java library, allowing reductions in code size of 10:1 or more for much of the basic graphics functionality in the system by using natural functional expressions such as:

```
Viewport.getBoundary().alignCentreTo(controlRegion.withMargin(10).getTopCentre());
```

The overall goal in the Palimpsest development was only to create an exploratory prototype, not a deployable application. In particular, the target platform was not current tablet products, but the expected resources of a product occupying this market segment in approximately 10 years time. Execution speed and memory requirements of the current prototype are well within this anticipated envelope. Despite the relatively free use of memory in full resolution image buffer caches for every layer, the current prototype runs comfortably with a 1GB Java heap. This allows creation of Palimpsest “applications” of around 10-30 active layers, of which a dozen or so may be encapsulated within collections. The resulting applications provide simple

animation functionality, and run for hours with no performance degradation.

Response to user actions offers smooth feedback to continuous interaction, except in cases where blurred shadows must be updated across the whole display image for each frame.

The development and experimental platform for this work has been a Macintosh MacBook Pro, running OS X version 10.7, with 2.2 GHz Intel Core i7 processor and 8GB 1333 MHz DDR3 memory. This is certainly outside the current range of the intended target platform - handheld touch-screen devices – but there is no reason to doubt that such devices will have specifications in this order within 10 years. A further constraint is that the programming model of the current iOS and Android canvas classes is less well suited to the implementation approach taken than the Java Graphics2D package. Market-driven developments for hand-held devices currently prioritise 3D rendering primitives, rather than alpha compositing of a large number of flat planes. Nevertheless, such capabilities seem likely to make the transition from desktop GUI libraries to handheld devices. Furthermore, the Palimpsest execution model is well suited to multi-core processes, as much of the rendering calculation required for individual layers can be performed relatively independently.

#### **4. Potential Applications**

The distinctive interaction style of Palimpsest has been inspired by the working processes of visual artists, but is also more broadly applicable. The motivating example given in the introduction to this paper was relatively trivial, expressed in terms of Photoshop macro recording. However, the general purpose computational functions of Palimpsest are intended to support larger-scale applications. As noted, the processor and memory requirements for running a practical Palimpsest program are larger than currently available on a high-end laptop, and 20-50 times greater than the target platform of typical consumer touch-screen tablets. As increased processing and memory capacities do become available, the following are examples of applications that would be well-suited to the Palimpsest model:

*1. Visual specification and control of real-time musical parameters.* There are a huge range of musical composition applications and games, intended for users without musical training, that offer novel mappings between visual variables such as location,

shape or colour, and musical attributes such as pitch, key signature, tempo etc (cite Stead, MelodyMorph). Each application of this kind uses different mechanisms to define song structures, alignment of multiple parts and other logical relations. Palimpsest could be used by end-users to invent their own musical notation, use it to control music synthesis, and construct their own musical composition applications involving any kind of structure or mapping they can imagine. The real-time performance of the current Palimpsest prototype is close to being acceptable for this application – an increase in processor speed of 5-10 times would make it a powerful authoring environment for new music synthesisers (using a real-time audio backend such as SuperCollider to generate the audio waveforms).

*2. Interactive data visualization.* There are many applications that provide partial mappings between numerical data and conventional chart displays, for example in spreadsheet applications. Users with more programming ability can customize and control a wider range of visualisation parameters with packages such as R or GnuPlot. And libraries such as D3 provide support for scripted control of data visualisation. However, Palimpsest makes it possible to specify any combination of mappings to visual elements, for example as recently demonstrated by Bret Victor (2011) in a demonstration of interactive textual programming. Data exploration with Palimpsest could be carried out completely in the visual domain by defining individual geometric elements in relation to imported data values, aggregating multiple pieces into a visual unit, encapsulating this as a Palimpsest layer stack, and then instantiating it over a data table to construct a display such as Tufte's visual multiples (1983). By comparison to the current Palimpsest prototype, an increase in processor speed of 50-100 times would support the use of Palimpsest as a powerful general purpose exploratory data visualization environment.

*3. Parametric design of geometric CAD models.* Powerful CAD systems allow the design parameters of products or buildings to be expressed in terms of geometric constraints, allowing designers to explore different options by manipulating a relatively small number of parameters that might result in different shapes or proportions of the overall building. These systems also allow multiple designs to be generated from a single abstract design model, with automatic generation of sets of detailed parts and drawings for each variation. Past experiments have represented constraint relations as visual networks (Aish 2000) or scripts (Aish 2011). In principle,

the Palimpsest interaction style could be used to define constraints on individual solid elements, encapsulated as layer stacks that defined their visual appearance and behaviour in the same context. By comparison to the current Palimpsest prototype, an increase in processor speed of 500-1000 times would support the functionality of a simple constraint-based parametric design environment.

## **5. Evaluation**

The current implementation of Palimpsest is fully functional, and includes all features that have been described and illustrated in the previous sections, although subject to the performance limitations already noted. However, this implementation can only be regarded as a concept demonstrator, rather than a functional product. In contrast, established products such as Photoshop include far greater incidental complexity (for example supporting a large number of import/export formats, standard image filters and conversions and so on). Established products have also become sufficiently robust that artists are able to develop skills and creative goals over years of practice. As a result of these two factors (incidental complexity and robustness), it is not feasible to evaluate the capabilities of a technical concept demonstrator in direct comparison to established products such as Photoshop.

Evaluation of Palimpsest has therefore followed a strategy developed for conceptual evaluation of novel interactive end-user programming paradigms. The "Champagne Prototyping" technique was originally developed to evaluate a functional programming extension to the Excel spreadsheet (Blackwell et al 2004). It uses supervised tutorials and demonstrations in order to evaluate the comprehensibility and perceived utility of novel programmable functionality within a specific context of use.

Two formal studies have been carried out in variants of this method – one controlled laboratory study, in which a range of users were observed as they worked systematically through a Palimpsest tutorial, and one field study, in which members of the core target user group of professional visual artists were interviewed in their own working context. In addition to these formal studies, Palimpsest has also been evaluated in experimental performance contexts, by comparison to live music coding languages.

### **5.1. Experimental study**

The goal of this study (reported in detail by Blackwell & Charalampidis (2013)) was to assess the extent to which Palimpsest was understandable and usable by individuals from a wide range of arts and technical backgrounds. 10 participants were recruited, all graduate students at the University of Cambridge, but enrolled on a variety of arts and scientific courses. Their degree of comfort with end-user programming was assessed using the computing self-efficacy questionnaire previously applied by Beckwith and others in end-user programming research (Beckwith et al 2006). Self-efficacy is a concept widely used in educational psychology to identify those whose confidence of their own ability in a particular subject is a good predictor of their performance when learning that subject. Participants also completed a similar questionnaire to assess their self-efficacy in visual arts (Hickman & Lord 2010).

All participants worked through a structured tutorial of Palimpsest functions, in the following order:

- Layers and the stack, the current layer and reordering layers
- Values and viewports, slider controls
- Images and masking
- Other value types: point, direction, rate and count
- Binding new values and value type conversions
- Path animations
- Collapsing layers into a collection
- Using the menu layer to access further functionality

As in previous uses of the Champagne Prototyping method, free-report data was collected and analysed in terms of Cognitive Dimensions of Notations framework in order to identify potential future usability problems. Quantitative measures included proportion of the tutorial voluntarily completed (one participant insisted on extending the experimental session to learn about every feature of the system), period of time spent in free exploration, favourable assessment of usability, and Likert-scale assessments of utility and enjoyability.

This study found that, although those with high computing self-efficacy (and also programming experience) understood the operating principles of Palimpsest, they did not perceive it as a practical alternative to conventional programming languages, unless they also had an interest in the visual arts. Those who had high self-efficacy in visual arts, but low computing self-efficacy, found Palimpsest complex and difficult

to understand. Those users who were most enthusiastic and engaged with the system had high self-efficacy in *both* visual arts and computing.

The usability issues observed in this study resulted mostly from the novelty of the interaction paradigm, and many are resolved simply by watching an expert user operate the system. These novel interaction elements include the manipulation of layers in the stack, and use of drag and drop to define constraints. Some elements of the design represent usability compromises because they have been created explicitly to enable more sophisticated higher-order programming operations (for example, the menu itself appears as a layer, allowing buttons and sub-menus to be invoked from other layers – usability would be improved if a conventional drop-down application menu were used instead).

## 5.2. Field study

In order to better understand the potential applications of Palimpsest in a professional visual arts context, a second study was conducted in which artists were interviewed in their own studios, followed by a variant of Champagne Prototyping in which the demonstration of Palimpsest capabilities used visual material that the artist had provided from their own archives (Williams 2014). Four artists were recruited, all professionals with a wide range of teaching, commission and exhibition experience. As before, the computing self-efficacy questionnaire was used to assess their prior familiarity and comfort with programming concepts – this ranged from extensive, to none at all.

In contrast to the controlled experimental study, the opportunity to experiment with their own visual material, and in their own working context, was far more effective in exploring the potential application of Palimpsest to their work. In future studies, it would be even more useful to assess self-efficacy in advance, and adapt the demonstration to individual confidence (for example, one participant used Photoshop very extensively, and initially perceived Palimpsest as a simplified version of Photoshop, which although attractive for its relative simplicity and usability, obscured the potential for programmable behaviour). These participants were also far more enthusiastic about the potential for experimentation, and would have appreciated a more extensive period of time to play with the system by themselves before the interview.

Nevertheless, this study offered a useful complement to the controlled laboratory study. Starting with a demonstration of the basic interaction paradigm meant that all participants understood the operation of layers and viewports. A more in-depth semi-structured interview also provided further insight with regard to usability issues and user motivations for using a system like this.

With regard to user motivation, not all artists are interested in using computers for purposes of creative experimentation. On the contrary, some of those in our sample associated computers with unwelcome bureaucratic aspects of their professional life, such as preparing business accounts, maintaining a website or sending email. Among these professional artists, this seems to be a more significant factor than the simple question of self-efficacy. Some are competent at using computers, but do not necessarily want to do so in a creative context.

With regard to usability, as with the controlled experiment, participants in this study commented on the relatively “clunky” appearance of the menu layer, a deliberate trade-off to support programmatic invocation of menu functions. However, a usability problem regularly raised in this study that had not been observed in the controlled experiment was lack of an undo operation. In comparison to direct manipulation environments such as Photoshop, using Palimpsest for basic image editing tasks highlighted the lack of a straightforward undo operation. This restriction is a fairly common drawback of programming by demonstration systems, for example as seen in Kahn’s *ToonTalk* (1996). The difficulty of implementing undo operations in this kind of system is in part a consequence of the complexity in unwinding an arbitrary execution. Simple operations in Palimpsest can usually be undone simply by removing a layer from the stack, or following the link to a layer removed from the stack.

### 5.3. Live performance

As a complementary evaluation perspective to these more conventional user studies, it is informative to consider Palimpsest in the context of “live coding” languages that are created by their authors for use in performance situations (Collins et al 2003) such as Aaron’s *Overtone* (2011), Sorensen’s *Impromptu* (2005) or Magnusson’s *ixi lang* (2010). These live coding languages are usually conventional text languages (e.g. Aaron is committed to use of ASCII tools (Aaron et al 2011)), although McLean’s

*AcidSketch* (McLean et al 2010) offers an example of an executable visual formalism used in performance. Live coders regularly implement languages for their own use, and even modify these in front of the audience. Conventional productivity and usability measures are only of limited relevance in this context – the main goal is to offer a novel experience to the audience, rather than to deliver a functional piece of software. As a result, some aspects of the language may be intentionally obscure or playful.

The author has given Palimpsest “performances” to a wide range of audiences. These have included audiences at live coding meetings and software research conferences, as well as more general audiences interested in the digital arts. As with some live coded music, these performances are often accompanied by a short talk placing the performance in the context of a research question or artistic intention. In this kind of context, the performance can be compared to a software demo, in which software research results are accompanied by a demonstration of the working software.

However, Palimpsest has also been used as a pure performance platform at a small number of events, in collaboration with live coding musician and researcher Sam Aaron. In one of these, live mixing and animation of images in Palimpsest created a visual projection improvised from a common theme shared with the musical improvisation (the theme was the song *Red Right Hand*, by Nick Cave, whose lyrics include vivid imagery). Although motivated by the popular nightclub combination of DJ + VJ (video jockey), this type of experimental fine art performance is more typical of the work of audio/video live coding performance duos such as *slub* (Alex McLean and Dave Griffiths (Armitage 2009)) or *klipp av* (Nick Collins and Fredrik Olofsson (2006)).

In another live improvisation with Sam Aaron, an extension Palimpsest layer type was created to send music control parameters via a socket to Aaron’s Overtone music synthesis language. Although Palimpsest itself does not run sufficiently fast to generate music, Overtone provided a real-time backend implementation based on the Palimpsest parameter specification. The parameters were generated from one button and one slider that could be bound via viewports to any dynamic event or animated value within a Palimpsest program, for example allowing a moving melodic profile to be defined in terms of a freehand ink shape. The audio output resulting from these



control parameters was defined on the fly in live coding by Aaron, including processed sound captured from violinist Tim Regan. This performance was in the context of a live improvisation free jazz ensemble, including both acoustic and electro-acoustic experimental instruments.

As with other live coding systems, functionality to support performance ideas continues to evolve in response to performance experiences. Some examples of specifically performance-oriented features that have emerged from experimentation include:

- all Palimpsest data values provide a starting point for creative exploration, rather than predictable behavior. For example, whenever a new value layer (e.g. a colour) is created, it is initialized to a random value – rather the same value last used (as in Photoshop) or a range extreme (as in most programming languages).
- fragments of the “editing” process can be captured and replayed within the “execution” process, for example via an interaction recorder that captures a dynamic sequence of interactions on another layer (an ink layer, for example), and replays them at variable rates.
- a snapshot layer preserves the current screen contents. This can be recorded either in exhibition mode, or with all the technical apparatus of Palimpsest visible, based on the observation that artists often choose to appropriate surface aspects of a technical context into their artwork.

As artistic rather than scientific experiments, these performances are useful indications of the potential value of Palimpsest, rather than measurable outcomes. Nevertheless, works created by Palimpsest have been enjoyed by audiences of some hundreds, and further invitations continue at the time of writing.

## **6. Related work**

The introduction to this paper described the operation of Palimpsest by analogy to image editing tools such as Photoshop. Users and audiences coming to Palimpsest from an arts background recognise this resemblance through the graphical user interface metaphor of superimposed layers. However, the technical capabilities and behaviour of the system have been influenced by prior research in the design of visual

programming languages, rather than image editing tools. In the current technology generation, graphical user interfaces and visual programming languages are considered to be separate fields of enquiry. But this research intentionally combines the two, by reference to the earlier history of systems in which visual programming languages also introduced new user interface metaphors.

The first of these is Sutherland's Sketchpad (1963), often recognised as both the first graphical user interface, and the first visual programming language. As with Palimpsest, the computational behaviour of Sketchpad was expressed in terms of constraints between the visual elements of drawings. As with Palimpsest, the drawings were not necessarily static (as in later drawing programs), but could potentially be animated by "running" the Sketchpad program - Sutherland's thesis ends with the description of a simple animation, in which a drawing of a girl's face is made to wink.

Kay's early work leading to the Smalltalk language was directly influenced by Sketchpad (Kay 1996), and by the recognition that the individual visual elements could be described by analogy to the independent behaviours of elements in a simulation language. Object-orientation thus has an origin in the composition of visual representations. The user interaction elements that were invented to interact with object data structures within a bitmapped display included windows, dialogs, icons and many other elements of the modern user interface. The key Smalltalk design principle that "everything is an object", allowing this uniform interaction style, is an example of how programming language innovation can drive user interface innovation. In Palimpsest, a similarly radical design philosophy started from the conjecture that "everything is a layer" would allow basic computational elements to be combined in a purely visual manner.

Smith's Pygmalion system (1977) made use of the Smalltalk bitmap display to extend the expressive power of Sketchpad, moving beyond constraint and type relations to explicitly support recursion. However the expressive power of Pygmalion was limited by comparison to Smalltalk itself, largely because of the self-imposed constraint of using only drawn images rather than symbolic strings to construct the concrete syntax. Following in this line, Furnas's BitPict (1991) explored the extent to which the pixel map itself can be used to express computation through local transformation rules.

BitPict was originally presented as a potential platform for prototyping novel user interface behaviours, although it has probably had more influence on end-user programming languages that used graphical re-write rules, such as Repenning's *AgentSheets* (Repenning & Sumner 1995) and Smith and Cypher's *KidSim/Cocoa* (1994). Palimpsest uses constraints, rather than re-write rules, as its primary computational formalism. Nevertheless, it is a conscious attempt to explore the purely-visual ambitions of these languages.

Among graphical constraint languages, one of the most extensive early examples was Borning's ThingLab (1981), also developed in SmallTalk, and a significant influence on SmallTalk successors such as *Squeak eToys* and *Scratch*. ThingLab returned to the simulation perspective of Simula, Sketchpad and SmallTalk, in which the constraints express physical laws and object properties. A later family of applications is the use of constraints to define 2D or 3D geometry in parametric computer-aided design (CAD) systems. A number of these systems include graphical languages for the specification of geometric constraints, for example Aish's *Custom Objects* (2000). In Palimpsest, all data structures are replaced by constraint relations, to an extent that the multiple controls on a layer may seem like parameters to a function. However, the continuous live execution model of Palimpsest means that every "assignment" of a value to one of these parameters is in fact a constraint binding.

In Palimpsest, the cache status of image buffers for each layer in the stack is determined via the constraint network, with the layer only re-rendered when the user is interacting with it, or when updates are propagated from other layers. This constraint architecture, with all parameter and binding values implemented via indirection operators (the viewports) that can be inspected and modified by users, resembles the pointer constraints employed in Myers' *Garnet* (Vander Zanden et al 1994). However, where Garnet is implemented in a relatively conventional programming environment, with the constraints explicitly specified, constraints in Palimpsest are often created implicitly. Where the behavior of a layer is determined by a historic dependency (for example, the creation of a masked image layer requires an image), Palimpsest searches down the stack to find the closest layer of the appropriate type. This is analogous to *implicit parameters* in the Scala language, which provides a degree of fluency that is valuable for rapid exploration.

## 7. Discussion

The theoretical objective of this research has been to explore new relationships between direct manipulation and programmable abstraction. Most programming languages express abstractions in textual language, even where the program will be processing images. Direct manipulation systems such as conventional image editing applications make it more straightforward for users to modify images, but do not support abstraction. Visual languages offer the potential for more abstract interaction with images, and Palimpsest is an extreme experiment in that direction.

A general theoretical approach to the relationship between direct manipulation and abstraction is provided by the Attention Investment model of abstraction use (Blackwell 2002). Although this is a cognitive model oriented toward design analysis, it does not directly offer design recommendations. Previous work by Wilson, Burnett et al. (2003) has operationalized the attention investment model in a specific design strategy for end-user debugging that they describe as “Surprise, Explain Reward”. One objective of the current research was to identify further concrete design guidelines of this kind.

A running theme in this paper has been the need for smooth transitions between direct manipulation and the definition of abstract behavior in Palimpsest. This both supports exploratory artistic practices, and also avoids the negative consequences of the Cognitive Dimension of *abstraction hunger*, as exhibited by many programming languages and tools. In Palimpsest there are two transitions in the level of abstraction provided during system exploration. The first is between direct manipulation of an image control, and indirect manipulation via a value layer. The second transition is the composition of the behaviors created using value layers and viewports, by collapsing into collections, by copying, or by modifying viewports and commands with indirection layers.

In homage to the Surprise, Explain, Reward design strategy developed by Burnett’s group, this approach to the transition between direct manipulation and programming functions can be described as Manipulate, Automate, Compose. The user is able to achieve useful results, and also become familiar with the operation of the system, through direct *Manipulation* that provides results of value. The notational devices by which the direct manipulation is expressed can then be used as a mechanism to

*Automate* them, where the machine carries out actions on the user's behalf. Finally, all of the functions that the user interacts with in these ways can be *Composed* into more abstract combinations, potentially integrated with other powerful and/or complex computational functions.

The different classes of potential Palimpsest applications suggest different user populations for whom the potential for increasingly abstract interactions might be appropriate in future. These might include musical improvisers, data analysts or architects. In each case, Palimpsest could support the expression of computational behaviour in the same context as interactive visualisations, without needing to resort to text when expressing abstractions. In more creative task contexts, the availability of an explicit action history may provide a more powerful view than typical versioning functionality, in that all operations on an image can be retrieved, removed or modified.

Users who are already engaged in digital arts find the Palimpsest paradigm intriguing, and the prototype engaging. Although the prototype is not yet suited to serious application development, they enjoy using it, and would be curious to explore its potential. Users who are familiar with a range of programming languages, but not engaged in the arts, appreciate the novelty of the Palimpsest paradigm but do not view it as a practical programming language syntax for general purpose use. It offers relatively extreme trade-off choices on a number of Cognitive Dimensions – for example, representation of a simple integer value occupies nearly a full screen, which is a dramatic illustration of *diffuseness*.

Palimpsest is clearly unusual as a programming language, but at this stage of development, it is not clear what other applications it might have. Users who work in the arts, but are not programmers, find it difficult to assess the potential applications of Palimpsest. If they do not use digital tools in their artistic practice, they view computers only as business tools (e.g. “could I use it to update my web page?”). Without clear evidence of a potential end-product, they see no reason to engage with a tool such as this, unless they wish to explore digital representations for their own sake.

In the digital arts and cultural studies of computing, there is considerable interest in *esoteric* programming languages such as Mondrian, Befunge and Brainfuck (Cox & McLean 2013) that are created as artistic or conceptual explorations rather than being intended as practical software engineering tools. There is some intersection between

these languages, and the languages developed for live coding performance, where the audience view both the source code and the resulting artefact. Languages such as Magnusson's *ixi lang* (2010) and McLean's *Tidal* (2010), while not immediately comprehensible to audiences, provide a visible relationship between construction and execution that offers a puzzle to be interpreted rather than a straightforward expression of an algorithm. Palimpsest is appreciated in this kind of context (for example, in an invited session at the 2012 Psychology of Programming Interest Group conference, where McLean and Magnusson gave live coded musical performances alongside a Palimpsest demonstration).

Current work is extending Palimpsest for use in this kind of application, for example via improved musical and visual integration with Aaron's Overtone live coding language (2011). However, future research will return to exploring more general image processing applications, as they become practical through greater memory capacity in graphics cards and hardware support for alpha blending. The most significant potential for extension of the Palimpsest paradigm is likely to arise from the availability of more powerful graphics libraries for functional reactive programming languages. Palimpsest could be reimplemented in such a language far more straightforwardly and robustly than the current Java Graphics2D implementation.

## **8. Conclusion**

This paper has presented Palimpsest, a novel purely-visual language for exploratory programming. It offers a new paradigm for visual languages, based on a spreadsheet-like constraint specification mechanism that is integrated with image composition functionality. Based on this novel interaction mechanism, a powerful range of computational functions can be supported, including data types that can both be directly manipulated and provide a basis for abstract composition of functionality.

At present, Palimpsest has been applied and evaluated in arts-related contexts, where the live execution is appropriate to performance situations, and the potential for exploration of image processing operations offers creative potential for the visual arts. However, the underlying interaction metaphor may also offer potential to support scripting or automation applications in future generations of keyboardless and

touchscreen devices. The experimental implementation that has been described in this paper currently requires significantly greater computational resources than those presently available on such devices, but reimplementing on more powerful hardware will allow such applications to be investigated.

## References

Aaron, S., Blackwell, A.F., Hoadley, R. & Regan, T. (2011). A principled approach to developing new languages for live coding. In *Proceedings of New Interfaces for Musical Expression*.

Aish, R. (2000). Custom Objects: A model-oriented end-user programming environment. *Workshop on Visual Languages for End-User and Domain-Specific Programming*. University of Washington.

Aish, R. (2011). DesignScript: Origins, Explanation, Illustration. *Proceedings of the Design Modelling Symposium*, pp 1-8.

Armitage, T. (2009). Making music with live computer code. *Wired UK*, 24 September 2009.

Beckwith, L., Kissinger, C., Burnett, B., Wiedenbeck, S., Lawrance, J., Blackwell, A. and Cook, C. (2006). Tinkering and gender in end-user programmers' debugging. In *Proceedings of CHI 2006*, pp. 231-240.

Blackwell, A.F., McLean, A., Noble, J. and Rohrer, J. (2014). Collaboration and learning through live coding. *Dagstuhl Reports* 3(9), 130-168. Edited in cooperation with Jochen Arne Otto.

Blackwell, A.F. (2002). First steps in programming: A rationale for Attention Investment models. In *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments*, pp. 2-10.

Blackwell, A.F., Burnett, M.M. and Peyton Jones, S. (2004). Champagne Prototyping: A research technique for early evaluation of complex end-user programming systems. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 47-54.

- Blackwell, A.F. and Charalampidis, I. (2013). *Practice-led design and evaluation of a live visual constraint language*. University of Cambridge Computer Laboratory Technical Report UCAM-CL-TR-883.
- Borning, A. (1981). The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Trans. Program. Lang. Syst.* 3, 4 (October 1981), 353-387.
- Citrin, W., Doherty, M. and Zorn, B. (1994). Formal semantics of control in a completely visual programming language. In *Proc. IEEE Symposium on Visual Languages*. 1994. St. Louis, 208-215.
- Collins, N., McLean, A., Rohrhuber, J. and Ward, A. (2003). Live Coding in Laptop Performance. *Organized Sound* 8, 321–330.
- Collins, N. and Olofsson, F. (2006). klipp av: Live Algorithmic Splicing and Audiovisual Event Capture. *Computer Music Journal* 30(2), 8-18.
- Cox, G. and McLean, A. (2013). *Speaking Code: Coding as Aesthetic and Political Expression*. MIT Press.
- Dillon, S. (2007). *The Palimpsest: Literature, criticism, theory*. Continuum.
- Furnas, G.W. (1991). New Graphical Reasoning Models for Understanding Graphical Interfaces, *Human Factors in Computing Systems CHI '91 Conference Proceedings*, New Orleans, April 28 - May 2, 1991, 71-78.
- Green, T.R.G. & Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' approach. *Journal of Visual Languages and Computing*, 7,131-174.
- Hickman, R. & Lord, S. (2010). An examination of adolescents' self-efficacy, engagement and achievement in representational drawing. *Australian Art Education* 32 (2),73-85
- Kahn, K. (1996) ToonTalk™—An animated programming environment for children. *Journal of Visual Languages and Computing* 7 (2): 197–217.



- Kay, A. (1996). The early history of Smalltalk. In *History of Programming Languages II*, T.J. Bergin, Jr. and R.G. Gibson, Jr., eds. ACM, New York. 511--598.
- Kurlander, D. and Feiner, S. (1990). Editable graphical histories. In E.P. Glinert (ed.). *Visual Programming Environments: Applications and Issues*. IEEE Press, Los Alamitos, CA. pp. 416-423.
- Lieberman, H. (Ed.). (2001). *Your Wish Is My Command: Programming by Example*. San Francisco: Morgan-Kaufmann.
- McLean, A., Griffiths, D., Collins, N., and Wiggins, G. (2010). Visualisation of Live Code. In *Electronic Visualisation and the Arts London 2010*. Available online at <http://yaxu.org/visualisation-of-live-code/> (accessed 22 Mar 2014).
- McLean, A. and Wiggins, G. (2010). Tidal – Pattern Language for the Live Coding of Music. In *Proceedings of the 7th Sound and Music Computing conference*.
- Magnusson, T. (2010). ixi lang: a constraint system for live coding. In *Proceedings of the 16th International Symposium on Electronic Art*, pp. 198-200.
- Repenning, A., & Sumner, T. (1995). Agentsheets: A medium for creating domain-oriented visual languages. *IEEE Computer*, 28(3), 17-25.
- Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., Kafai, Y. (2009). Scratch: Programming for All. *Communications of the ACM*, November 2009.
- Schachman, T. (2011). *Recursive Drawing*. Online application available at <http://recursivedrawing.com/draw.html> [last accessed 30 August 2012]
- Shneiderman, B. (1983). Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer* 16, 8 (August 1983), 57-69.
- Smith, D.C. (1977). *PYGMALION: A Computer Program to Model and Stimulate Creative Thinking*. Birkhäuser, Basel, Switzerland.
- Smith, D.C., Cypher, A. and Spohrer, J. (1994). KidSim: programming agents without a programming language. *Communications of the ACM* 37 (7), 54-67.

- Smith, R.B. (1986). Experiences with the alternate reality kit: an example of the tension between literalism and magic. *SIGCHI Bull.* 17, SI (May 1986), 61-67.
- Sorensen, A. (2005). Impromptu: An interactive programming environment for composition and performance. *Proc. Australasian Computer Music Conference*, Brisbane: ACMA, pp. 149–153.
- Sutherland, I.E. (1963/2003). *Sketchpad, A Man-Machine Graphical Communication System*. PhD Thesis at Massachusetts Institute of Technology, online version and editors' introduction by A.F. Blackwell & K. Rodden. Technical Report 574. Cambridge University Computer Laboratory
- Tanimoto, S.L. (2013). A Perspective on the Evolution of Live Programming. In *Proc. First International Workshop on Live Programming*, (in association with ICSE 2013), pp. 31-34 Available online at <http://www.cs.washington.edu/ole/Liveness2013.pdf> (last accessed 22 March 2014).
- Tufte, E.R. (1983). *The visual display of quantitative information*. Graphics Press.
- Vander Zanden, B., Myers, B. A., Giuse, D., and Szekely, P., Integrating Pointer Variables into One- Way Constraint Models, *ACM Transactions on Computer Human Interaction*, 1, 2, 161-213, 1994.
- Victor, B. (2011). *Dynamic Drawing*. Blog entry dated Mar 3 2011. Available online at <http://worrydream.com/#!/DynamicPicturesMotivation> [last accessed 26 Aug 2012]
- Wellner, P. (1993). Interacting with paper on the DigitalDesk. *Commun. ACM* 36, 7 (July 1993), 87-96.
- Williams, M. (2014). *Evaluation of a live visual constraint language with professional artists*.
- Wilson, A. Burnett, M., Beckwith, L., Granatir, O., Casburn, L., Cook, C., Durham, M., and Rothermel, G. (2003). Harnessing curiosity to increase correctness in end-user programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '03)*. ACM, New York, NY, USA, 305-312.

## Appendix A – Tutorial Example

This tutorial includes thumbnail images taken from a narrated video demonstrating the application example described in section 1.1. The full video is provided in the supplementary materials supporting this paper.



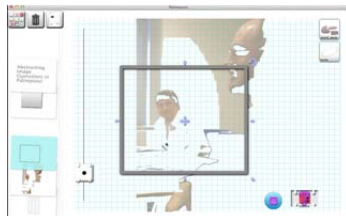
When an image is loaded into Palimpsest, it appears as a new layer



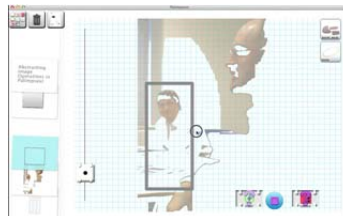
Clicking the “cut-out” button removes the background, so that layers beneath are visible.



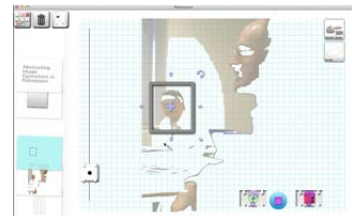
The drawing menu provides a set of basic geometric shapes



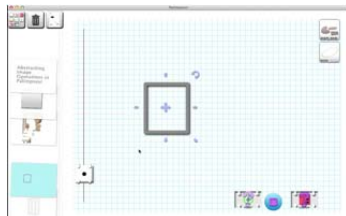
Clicking the rectangle button creates a new shape layer on top of the image layer



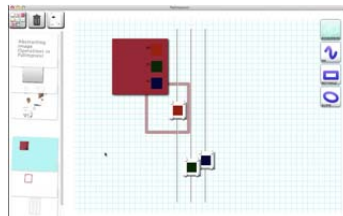
Layer contents are moved and resized using standard drag handles



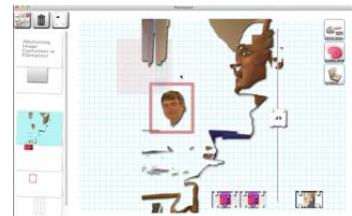
The stack of layers can be reordered by dragging the thumbnails at the left



The rectangle layer is now below the image layer



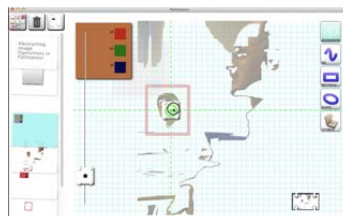
Clicking the empty color viewport creates a new color layer with RGB sliders



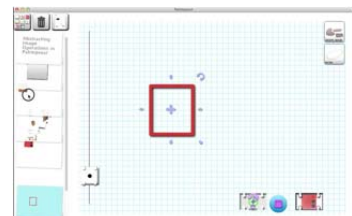
The color of the rectangular frame behind the image is now defined by that color layer



The menu of value layer types includes a color sampler for dynamic color values



The color sample layer obtains a color from a specified point within an image layer



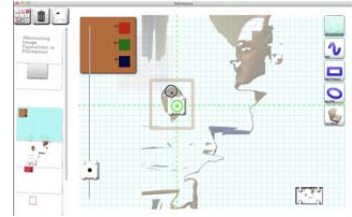
Now we can return to the rectangle layer, and bind its color to the sampled value



The color sample value is dragged across the screen from the layer stack



A green highlight shows that the value can be dropped onto the color viewport



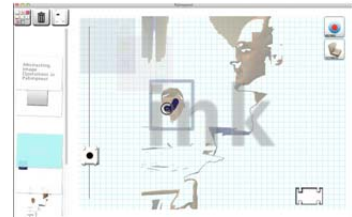
Now the color of the rectangle is bound to the sample value



When the sample point is dragged to a different location, the color changes dynamically



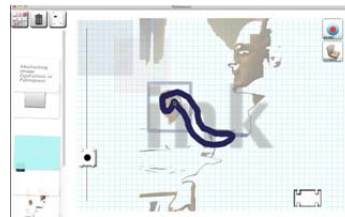
A dynamic path can be defined for the sample point by creating another shape layer



An ink layer has been created, allowing the user to draw a freehand path.



The user wants various color over the face, followed by the color of the shirt



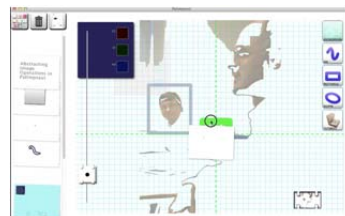
Closing the ink path provides a looped animation



Clicking the path animation button creates a new layer with a point value that moves over the shape



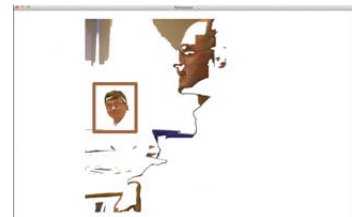
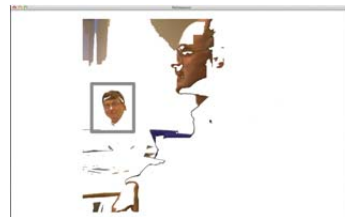
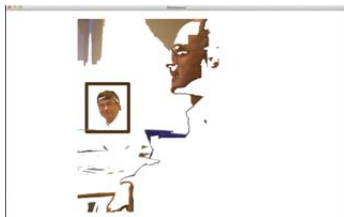
The layer representing the moving point value is dragged into the main area



The green highlight shows that the moving value point can be bound to the color sample position.



The final result is a frame that changes its color continuously, based on a path sampled from the framed image



## Appendix B – Full Palimpsest Tutorial

This appendix reproduces the full text of the built-in interactive tutorial included with the Palimpsest system. It is provided as a reference for the available layer types, and also gives a flavour of what it is like to interact with the system when working through this tutorial.

### *Introduction*

Welcome to Palimpsest, a sketchbook for playful exploration of powerful computer graphics.

A palimpsest is a picture with many layers.

Try making some new layers underneath this text by pressing the ellipse button below.

(then to continue this introduction, press NEXT)

### *Layers and Stack*

The main part of the screen shows the current layer (now it is this text). You can see through the current layer to layers stacked underneath.

The stack overview at the left shows all the layers. You can choose which one is current by clicking on it.

Try that now, then use the stack overview to come back to this text

### *Layer Order*

Note that that the main view looks "down" at the stack layers below the current one, while layers above it can't be seen.

The order of the layers can be changed by dragging them up and down in the stack overview at the left.

Try it now (then come back to this text).

### *Values*

Many layers have values that can be changed.

Values are on separate value layers, connected by "viewports" between layers.

Try clicking on the empty viewport below, to make a value layer that changes the colour of this text with sliders.

### *Slider Controls*

For convenience, some values can be changed on the same layer where they are used.

Try dragging the slider handle at the right up and down.

You can also convert a slider to a separate value layer by clicking on the handle (without dragging).

### *Images*

Image layers can be created by dragging image files (png, jpg or gif) into Palimpsest from the desktop or a folder.

You can also browse for images by pressing the LOAD IMAGE button.

### *Masks*

You can extract part of an image using a mask.

Masks can be made in any shape.

Try using the button at the top right of an ellipse layer to make a mask.

### *Applying Mask*

A value from one layer can be used in another by dragging it from the stack overview into the main view area. Try applying a mask to an image like this:

1. Click on an image layer in the stack overview at the left, to make it the current layer in the main view.
2. Drag a mask layer from the stack overview and drop it into the main view area.

### *All Value Types*

There are several kinds of value. You have already seen Colour, Amount, Image and Mask - others are:

Point (for places on the screen)

Direction (for movement)

Rate (for speed)

Count (for repetition)

Try them out using the buttons.

### *Viewports*

Wherever a viewport shows a value from another layer, it can be replaced.

Use the button on the left below to make another colour layer, then drag that layer out of the stack overview, and onto the colour value viewport on the right.

The viewport changes to show the new value - try changing it back to the original colour.

### *Hiding Layers*

You may have noticed that to reduce clutter, the original image layer was hidden, by removing it from the stack.

A viewport in the mask result shows the original layer. You can return the hidden layer to the stack by clicking on that viewport.

### *Animation Paths Intro*

A shape can be used to define the path of an animated point.

Make an ellipse, then use the PATH button at the right of the ellipse layer to create an animated path layer.

### *Value Conversions*

It is often useful to convert values of one kind into another.

For example, use the button at the top right of an animated path layer, to create an amount based on the height of the moving point.

Then go to a layer with an amount slider (e.g. a colour) and drag the new moving amount from the stack overview onto a slider handle instead of a viewport.

### *Collapsing into Collections*

Layers can be organised by collapsing sections of the stack into a collection layer.

Try it using the button below. You can return layers to the stack by going to the new collection layer, and clicking on the one you want.

### *Moving to the Trash*

Unwanted layers can be removed from the stack by clicking the trash-bin at the top left.

This creates a trash layer at the bottom of the stack.

You can retrieve deleted layers by going there, and clicking the one you want.

### *Shortcut Keys*

There are a few useful shortcut keys:

Up and down arrows move up and down the stack. Up/down with the shift key held down swaps the current layer with the one above or below it.

The delete key has the same effect as clicking the trash bin.

The space bar starts an exhibition mode with all controls hidden.

### *Menus*

Different kinds of layer can be created using buttons on a menu, accessed via the menu button at the top left of the Palimpsest window.

Try it now to see the available menu tabs - some buttons will already be familiar, while others are advanced functions to explore in future.

### *End of Basic Tutorial*

You have now seen the basic Palimpsest functions. To explore other types of layer, use the menu button.

Many layers also have buttons at the top right, suggesting possible ideas to explore next.

More advanced layer types also have a button to request a tutorial page with more advice.

### *Drag*

(Advanced) Basic shapes and images can be moved, resized or rotated by dragging handles.

The actual amounts by which the handles are dragged can be used as value layers - click on the viewport to see the layer for move (the centre point), rotate (a direction) or scale (an amount).

Drag a different value layer into the viewport (for example a moving point) to automate these actions.

### *Event Values*

(Advanced) All buttons, sliders and value viewports can be automated using Event layers.

To create an Event layer, right-click on a control - try it on the "NEXT" button below. If you drag an event layer onto a path, the event will be triggered when the animated point goes past it.

### *Advanced Introduction*

(Advanced) Reference entries for other layers.

The remaining tutorial pages cover advanced topics. Rather than reading through in order, it may be more convenient to return to individual pages when they are needed, using the TUTORIAL button that appears on the right hand side of many layers.

### *Paragraph*

(Advanced) The paragraph layer arranges text over multiple lines.

Different shapes can be defined with a mask.

Note that there are almost no editing functions. The best way to enter text is to create it in an external application, copy it, and use the paste button on the paragraph layer.

### *Ink*

The ink layer lets you build up a shape from freehand elements.

Individual ink strokes can be dragged around to change their position, or dragged out of the main viewing area into the stack overview to put them on a separate layer.

### *Spin*

(Advanced) The spinning direction turns clockwise or counter-clockwise at the specified rate.

If a specific distance (length) is required, create a normal direction value with the required distance using the direction viewport in the spinning direction layer.

### *Using Collections*

A collection combines its members into a single image - use a mask to include only some of them.

You can replace the whole stack with the contents of a collection, using the big blue button. To return the original stack, a backup copy is provided at the top of the new stack.

Note that you can view collections either as a stack, or arranged into a grid to see the contents more easily.

### *Sequences*

(Advanced) A sequence of points or amount values can be created in a single collection.

Use a count value to define how many values there will be in the collection.

Controls on the sequence layer specify the starting value, and the difference to be added between each member of the sequences.

### *Copying Layers*

(Advanced) There are some powerful facilities to copy collections of layers.

Drag a collection of values into a viewport on another layer, to make multiple copies having those values.

Alternatively, use the copy button to copy a single layer or collection, and draw a mask over any viewports that should have the same value as the original.

### *Mask Advanced*

(Advanced) Mask layers can be built up from multiple sources.



You can drag additional shapes onto a mask layer, and they will be added to the mask area.

You can also paint additional mask areas, or erase areas, by drawing directly on the mask.

### *Blur*

(Advanced) Blur layers provide a blurred version of another image, but can also be used as a soft-edged mask.

The standard blur is evenly textured. A textured blur can also be created by dragging a suitably textured image onto the blur layer - light and dark areas of that image define transparency variations.

### *Calculator*

The calculation layer takes an amount as an input, and turns it into a different value.

The central area shows the original amount. The dotted rectangle shows how this original value is adjusted - first by adding or subtracting height, then by scaling the width.

Try playing with the sliders to see what effect they have.

### *Sweep*

Use the sweep layer over a changing image, such as a moving line.

New versions of the image are drawn over older ones, which gradually fade.

The fade rate can be adjusted by dragging a rate value onto the sweep layer.

### *Making Points*

(Advanced) There are two ways of converting other layers to point values - using the centre point of any image or shape, or using the middle of two other points.

Use the viewing mode control to choose which (or all) of the alternative conversions is visible.

### *Making Amounts*

(Advanced) There are several ways of converting other layers to amount values - using the position of a point on the screen (either X or Y), using the size of a shape or image (either width or height), or using a direction (either angle or distance).

Use the viewing mode control to see all of the available conversions.

### *Making Directions*

(Advanced) There are several ways of converting other layers to direction values - using the path between two points, using the long axis of a shape or ink layer stroke, or using amounts for angle and distance.

Use the viewing mode control to choose which (or all) of the alternative conversions is visible.

### *Colour Sampler*

(Advanced) The colour sampler defines a colour value taken from a patch at a specific point in an image.

The size of the sample patch can be adjusted - the resulting value is the average colour across the patch.

### *Event*

(Advanced) The event value layer can be used to produce regular events, occurring at a specified rate.

When triggered, it repeats as many times as specified by the count value.

If the repeat mode button is turned on, this continues until repeat mode is turned off.

### *Cut Out*

The cut-out layer can be used to cut out a foreground picture from an even-coloured background.

Both light and dark backgrounds are recognised. The slider adjusts the amount of the picture remaining between light and dark.

Colour values can be specified to replace either the light or dark parts with an even colour. If no replacement colour is specified, those areas are left transparent.

### *Rubbing*

The rubbing layer lets you "rub" over an image to extract the parts you want.

It uses a hidden mask layer that can be viewed by clicking the mask viewport.

Extracted parts can be dragged onto the stack overview to make new layers.

### *Animation Paths Complete*

An animated point moves between locations along a path at a steady speed, controlled by the rate value.

The path can be defined either by a single shape, or by the outline of a mask.

If an event layer is dragged onto the path, that event will be triggered when the moving point passes the nearest path location to the event marker.

### *Mouse Tracker*

The mouse tracker can be used to make a Palimpsest display that reacts to user actions.

While active, the point value follows the mouse. Use the play button, keyboard Enter, or click to start and stop tracking.

The point can also be moved up and down with arrow keys. Hold down the option key for fine adjustment.

### *Recording*

The recording layer can be used to record all mouse actions in the layer underneath it, then play them back at an adjustable rate.

There is no way to make multiple recordings at present. Moving to other layers before stopping the recording can also have unpredictable effects.

### *Choice*

(Advanced) The choice layer can provide any kind of value, choosing between two alternatives of the same type (two colours, two shapes, two masks, two points etc).

To automate the selection, make a trigger by right-clicking on a control button that selects one alternative or swaps between them.

### *Comparison*

(Advanced) The comparison layer compares two amounts, and presents a mask based on the result.

If the amount on the right is smaller, then the mask will be empty. If it is larger, the mask value from the viewport is used.

The resulting mask can be used to make an image appear, or to activate members of a collection.

### *Dependency Graph*

(Advanced) The dependency graph can be used to review which layers use values from other layers.

The graph can quickly become crowded, so it may be helpful to collapse the layers of interest into a separate stack.

### *Annotation*

The annotation layer can be used to make notes about a value on another layer.

It will also pass on that value, controlled by a disable/enable button.

Sorry, but the text editing functions are useless. I recommend that you copy your text from outside Palimpsest, and use the paste button.

### *Trigger Commands*

(Advanced) Trigger layers can automate the behaviour of a button or slider.

The event viewport on the trigger layer can be used to coordinate or repeat multiple triggers.

As with other event layers, a trigger layer can be dragged from the stack onto an animated path. The trigger action will occur when the animated point passes the event marker.

### *Trash*

The trash collection can be used to store layers not required in the stack (even if some viewports still use them).

To clean up the trash collection, click the trash bin icon. The cursor then changes to deletion mode - any member of the collection that you click on will be deleted.

### *Multi Value Copy*

(Advanced) The multi-value copy can be used to make multiple copies of another layer, each using a different value from a collection.

Different buttons make either a single layer copy using the next value, multiple copies on the stack for each value, or multiple values placed into another collection.

### *Shuffle Stack*

The shuffle layer can be used to change the order of the stack.

The up and down buttons at the left move the current layer view up and down the stack (like up/down arrow keys).

The buttons at the right change the order by moving the layer underneath up or down the stack (like shift up/down).

The centre button turns exhibition mode on and off (like space bar).

### *Snapshot*

The snapshot layer creates a snapshot of the whole stack, as seen from under this layer.

It captures either the exhibition mode, or the user view of Palimpsest - including any controls on the layer below.

To reduce confusion with controls in the picture, the controls for saving are inside a grey border.

### *Recovered Stack*

This layer preserves the stack as a collection of layers.

You can replace the whole stack with this collection, using the big blue button.

To return the original stack, a backup copy is provided at the top of the new stack.

### *Viewport Status*

The viewport status layer lets you modify the way that one layer uses a value from another.

The connection between the layers can be temporarily disabled or enabled by clicking the cross/tick button. It can be cancelled altogether by clicking the trash icon.

### *End*

This is the end of the tutorial.

Other features of Palimpsest can be discovered by exploring the menus, and trying the layer types suggested at the top right of each current layer.

Enjoy playing with Palimpsest!

## **Appendix C: Capsule definitions of Cognitive Dimensions**

The Cognitive Dimensions of Notations framework was introduced in this journal by Green & Petre (1996). A special issue of this journal (Vol. 17 No. 4) presented papers reviewing applications of the framework over the subsequent 10 years.

The underlying concept of the Cognitive Dimensions framework is that the user is considered to be interacting with an information structure, which is composed of components and the relationships between them. The notation includes visual representations of components and relationships. The usability characteristics result from the interaction between the visual representation and the environment that is used to view, navigate, create and modify it.

The following definitions are based on those presented in Blackwell, A.F. and Green, T.R.G. (2003). Notational systems - the Cognitive Dimensions of Notations framework. In J.M. Carroll (Ed.) HCI Models, Theories and Frameworks: Toward a multidisciplinary science. San Francisco: Morgan Kaufmann, 103-134.

Abstraction	availability of abstraction mechanisms, ranging from <i>abstraction hunger</i> (abstractions are required) to <i>abstraction hating</i> (abstraction are not possible).
Closeness of mapping	correspondence between representation and application domain.
Consistency	similar semantics are expressed in similar syntactic forms.
Diffuseness	verbosity of language, for example in terms of screen real estate.
Error-proneness	the notation invites mistakes and the system gives little protection.
Hard mental operations	high demand on cognitive resources such as short term memory or maintenance of subgoal dependencies.
Hidden dependencies	important relationships between components are not visible.
Premature commitment	constraints on the order of doing things.
Progressive evaluation	work-to-date can be checked at any time.
Provisionality	degree of commitment to actions or marks.
Role-expressiveness	the purpose of a component within the overall structure is readily inferred.
Secondary notation	extra information can be expressed in means other than formal syntax.
Viscosity	resistance to change of the structure.
Visibility and juxtaposability	ability to view components easily.