Artistic resizing: a technique for rich scale-sensitive vector graphics

Pierre Dragicevic, Stéphane Chatty, David Thevenin, Jean-Luc Vinot

Artistic Resizing: A Technique For Rich Scale-Sensitive Vector Graphics

Pierre Dragicevic[1] *Stéphane Chatty*[1] *David Thevenin*[1] *Jean-Luc Vinot*[2]

[1]IntuiLab
Prologue 1, La Pyrénéenne
31672 Labège Cedex, France
{dragice,chatty,thevenin}@intuilab.com

[2]Direction Générale de l'Aviation Civile
DSNA/SDER, 7 avenue Edouard Belin
31055 Toulouse, France
jean-luc.vinot@aviation-civile.gouv.fr

ABSTRACT
When involved in the visual design of graphical user interfaces, graphic designers can do more than providing static graphics for programmers to incorporate into applications. We describe a technique that allows them to provide examples of graphical objects at various key sizes using their usual drawing tool, then let the system interpolate their resizing behavior. We relate this technique to current practices of graphic designers, provide examples of its use and describe the underlying inference algorithm. We show how the mathematical properties of the algorithm allows the system to be predictable and explain how it can be combined with more traditional layout mechanisms.

ACM Classification: H5.2 [Information Interfaces and Presentation] User Interfaces — GUI; D2.2 [Software Engineering] Design Tools and Techniques; D2.6 [Software Engineering] Programming Environments.

General terms: Design, Human Factors, Languages

Keywords: visual design, vector graphics, SVG, GUI tools, layout, resizing, constraints, interpolation

INTRODUCTION
Solutions are increasingly available for graphic designers to build the graphics of visual applications directly without having programmers translate their work into code. For instance, Macromedia Director allows designers to build simple multimedia applications. More recently IntuiKit proposes a solution to merge graphics with the more traditional software components of larger applications [6]. However, many graphic designers are willing to go further and take more control over the programming of user interfaces. When used in an interactive application, their graphics are often adapted to reflect context changes or data variations, and controlling these changes should be part of their job. In particular, they want to control the way the graphics are resized.

UIST'05, October 23-27, 2005, Seattle, Washington, USA.
Author's version.

A number of automated techniques are available to manage the layout of visual interfaces. They range from the simple rescaling to more complex constraint solving algorithms. However, most techniques are aimed at programmers and do not provide a fine enough level of control for designers. They do not take into account the fact that the eye is a subjective instrument and that deformations must be applied to objects in order to create an illusion of smooth resizing (see Figure 1). Consequently, designers have to choose between accepting a sub-optimal resizing of objects, providing graphics for every expected size, or ask programmers to produce of a specific resizing function for each object.

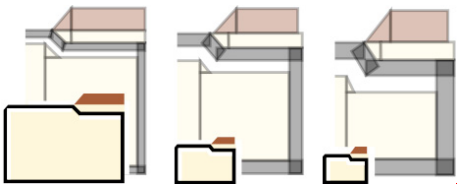

Figure 1: Artistic resizing: the same object at several key sizes. Details are magnified in the background.

Artistic resizing is a programming-by-example solution to this problem, aimed at graphic designers. It consists of providing several copies of an object that should have a specific resizing law. Each copy represents the object at a key size. The system then interpolates the variation of the object between these key sizes, when needed. It takes advantage of modern scalable vector graphics to provide pixel-precise control over the resizing of objects while preserving the appropriate level of generality. Furthermore, Artistic Resizing is a purely visual technique that relies on the use of current graphic design software. It thus empowers designers without disturbing current work practices. With Artistic Resizing added to IntuiKit, graphic designers and programmers can work in parallel as before but more work than usual is devoted to designers.

After summarizing the state of the art of layout and resizing techniques, we analyze how that activity fits in the practices of designers. We then show through examples how Artistic Resizing supports their needs. We then describe the internals of the system: example extraction and interpolation. We finally describe how the system can be generalised to other situations.

RELATED WORK

Advanced resizing behaviors in GUIs are often associated with layout management systems. Whereas in the "fixed layout" model, coordinates and sizes of elements do not evolve according to the container's dimensions, higher-level layout models allow the dynamic adaptation of graphical contents. We recall these models here, as well as the work on image interpolation in the fields of animation, typography and computer graphics.

Traditional Layout Management

The most popular approach to layout is the boxes and glue model, coming from the field of document publishing and widely adopted in GUI toolkits [22, 14]. Most of the power and ease of use of box layouts comes from the fact that boxes can be nested. But the model of containement hierarchy is also limitative in that it cannot be applied to rich graphics. For example, Scalable Vector Graphics (SVG) groups are used to describe objects that are expected to move together or that share graphical attributes, but rarely to express containment. In fact, containment is far less relevant in graphical design than composition and superposition of elements such as backgrounds, shadows and lights.

Constraints

Constraints have been a popular approach in research for expressing GUI layout [20, 10, 9]. They are more expressive and more declarative than boxes and glue models and do not have to rely on containement hierarchies. Constraints are easy to use when there are a few, but have proved to be complex to specify and maintain as their number increases [13, 3].

Some constraint-based GUI toolkits include graphical frontends for specifying layout. For example, an early but ingenious system allows specifying resizing behavior for windows by moving "attachment points" linked to object edges [5, 9]. A related approach relies on the struts and springs model [22]. Again, although these visual languages allow non-programmers to easily specify layouts, they can be difficult to use when visual elements are numerous.

Programming by Example

Most work on programming by visual examples focused on inferring the mappings between application data and its visual representation [18, 19, 17, 8]. Some systems like Peridot allow demonstrational layout specification by inferring simple geometrical constraints between visual elements [18]. But most PBE systems involve sophisticated inference engines and rely on user mediation for solving ambiguities, which is hardly compatible with the graphic designer's way of working.

An approach that successfully combines power and ease of use is the Chimera system [13], which can infer from multiple drawings geometrical invariants such as incidences, relative distances and slopes. Chimera's specification style is subtractive, i.e., all possible constraints are built first, then the users incrementally removes unwanted constraints by adding new examples. Resizing behaviors can be specified using a bounding box as part of the scene.

Although Chimera allows to visually express a wide range of resizing behaviors, it still shares some drawbacks with previous approaches. It can infer overconstrained or unintended rules, which require the user to draw more examples to prune away undesired constraints. Then, extensive search for invariants is a costly process, sensitive to combinatorial explosion. Our work is closely related to the Chimera system but is specifically targeted to resizing and better handles complex graphics. The technical differences between the two approaches will be highlighted later in this paper.

Bitmap Tiling

As far as we know, the only attempt to give graphic designers some control over resizing behaviors was by Hudson and al [11]. Designers provide bitmaps in nine parts that are resized differently in the GUI. Despite its simplicity, the 9-part tiling technique supports a resizing law commonly seen in resizable widgets. But tiling-based approaches rely on assumptions that place constraints on graphic designer's work processes. Moreover, they do not naturally extend to vector graphics which cannot easily be cut out into rectangular pieces.

Image Interpolation

Altough animations can be described procedurally or using constraints, key-frame interpolation is still considered as the most expressive method [26, 15]. In-betweening algorithms usually involve rigid interpolation of transformations [25]. Other schemes range from direct linear interpolation of coordinates to physically-based shape blending [24].

Parametric images are a generalization of animation. If most advanced techniques use domain-specific parametrizations such as joint angles for controlling anthropomorphic figures [23], a more general approach involves generating families of images by blending several examples. Ngo and al, for example, use piecewise linear interpolation and simplicial complexes to specify valid transitions between several images [21].

Parameterization is also supported in some font systems. Adobe's Multiple Masters [1] uses weighted linear interpolation between sample fonts, an algorithm similar to ours, to generate fonts variations according to "design axes" such as optical size, weight and style.

Another corpus of work comes from the field of image morphing, where bitmap images are transformed to retain geometric alignement between user-specified landmarks such as meshes or curves. Well-known algorithms are field morphing [4] and scattered data interpolation [27]. Most of them are tuned to give compelling effects on realistic photographs rather than to preserve simple geometrical properties.

Key-image interpolation has proved successful in several fields and our contribution shows that it can also be applied to the problem of GUI resizing. The technical issues are however different, especially when compared to animation. We will develop on these further in this paper and explain how a computationnaly costless variant of direct linear interpolation both solves the multivariable problem and allows describing most desired resizing behaviors.

HOW DESIGNERS PRODUCE VARIANTS

Despite being used to more stable media than interactive displays, most graphic designers are often faced with situations similar to resizing and layout. Indeed, preparing variants of their work is a common task for graphic designers. For instance, they learn to build fonts at different sizes during their typography courses. The same holds for layouts in layout courses. Another example is corporate design: in this case, they know that their work will be used by others on different media, at different sizes, on different color backgrounds and in different contexts. It is thus important to understand how they work with variants of their designs.

Scale-dependent Designs

The eye is a very subjective measurement system. To suit it and produce the desired result, resizing often has to be applied differently for different parts of a picture. In a font, two glyphs may need to have different sizes so as to appear similar, and the necessary adjustment will not change linearly with scale. Among three aligned lines in a picture, the most central has to be longer than the others to appear the same size, and here again the adjustment will not change linearly with size. The same holds for the space between words in a title, or between blocks in a layout. In addition to these factors, constraints have to be applied to ensure the readability of a picture : for instance, small parts of the picture will be reduced more slowly than the rest so that it stays visible.

Figure 2: A logotype prepared for different sizes. Details are magnified in the background and show the differences.

As an example, in Figure 2 we have scaled up (in pink and grey) two variants of the same logotype which are destined for different sizes. The actual use of each logotype is shown on top of it. What appears first is that the pink square has a bigger relative size when the logotype is prepared for a small scale: in order to keep the square visible, the designer had to reduce it less than the rest. The other effect is well known to font designers: the proportions of the Q-like glyph have to be changed depending on the scale of the glyph.

Working with the Eye, Explaining with Examples

Finding the design that works at a given size is a forward searching, heuristic activity. The laws that govern the relative sizes of object parts are often not linear, and actually unknown. Finding the appropriate proportion is a complex and iterative process of experimentation that stops when the designer's eye is satisfied. Designers are often unable to explain why the result works. That is why, when doing corporate design, they try as much as possible to provide samples for all situations thay can foresee: doing so, they avoid explaining how the result can be extrapolated. Furthermore, designers are used to a close experimentation-evaluation loop: the eye can evaluate quickly what the hand has attempted. Consequently, it is more efficient to experiment graphically then perform some reverse engineering on the result rather than code directly the resizing law, even for a designer with programming skills. Artistic Resizing builds on that understanding: it uses the examples produced by designers and does the reverse engineering for them.

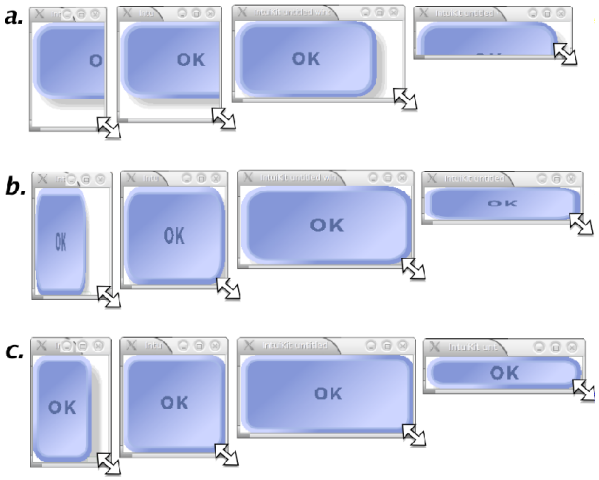

Figure 3: Three resizing strategies for a vector graphics button: fixed layout (a), naive scaling (b) and commonly expected resizing (c).

ARTISTIC RESIZING AT WORK

In this section, we introduce the main concepts behind Artistic Resizing through examples.

An Introductory Example

Figure 3 shows a simple dialog window with a SVG button inside. This dialog can have no resizing strategy at all (a) or take advantage of the scaling capabilities of vector graphics (b). But because naive scaling distorts the graphics, it may be more acceptable to preserve some graphical constraints such as text size, border width and rectangle roundings (c).

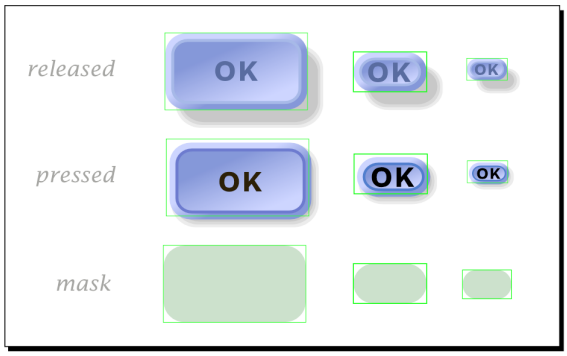

Figure 4: The SVG specification of Figure 3c, with the three button parts and their bounding boxes.

Figure 4 shows the SVG document drawn by the graphic designer to describe the button of Figure 3c. The way the pictures in the different rows of the figure can be combined to form an interactive button has been described in [6]. We will focus here on the Artistic Resizing that occurs between pictures of the same row. There are two different ways of explaining what happens:

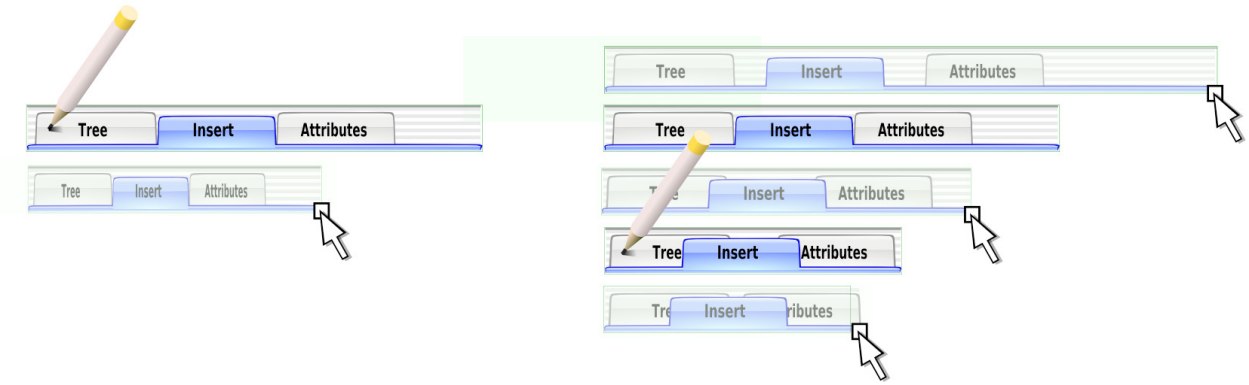

Figure 5: A possible scenario: on the top left, the graphic designer draws a graphical object and sees how it behaves when naively scaled (1). She then decides to draw a second example to specify another way of shrinking the object (2). The new resizing behavior can now be tested to see how it behaves between and outside the examples.

- *The constraint-solving view.* The first two examples specify what to do when the button is scaled down: the OK label must stay centered and always keep the same size, as well as the box roundings and the border width. Shadow location has also been constrained so that the button always appear to be at the same elevation. The third example specifies that when the button becomes so small it cannot contain the label any more, everything is scaled down.
- *The graphical design view.* Buttons with different sizes have been drawn in such a way that each of them appeared visually acceptable to the designer. Because designers cannot draw hundreds of them, some morphing algorithm will smoothly infer the intermediate sizes.

Both views are valid when drawing examples for Artistic Resizing. When the specification of Figure 4 is executed, constraints that were expected to be maintained are indeed maintained, even outside the specified size range. For example, the text stays centered, but a right or top alignement could be specified as well, as detailed later in this paper.

However, Artistic Resizing is not all about specifying invariants. A large palette of continuous changes can also be specified, such as those shown in Figures 1 and 2. For instance, the button designer can decide to slightly reduce the size of the label and make the rectangle sharper on the middle example. In this case, the changes would be smoothly interpolated and key sizes would remain unnoticed. While testing the result, the designer can choose to add more examples for the sizes she does not find appealing. In fact, Artistic Resizing encourages incremental design, as we will see on a sample scenario.

An Artistic Tab System

In this section, we use a scenario to shed some light on the process, as well as how more sophisticated behaviors can be described. Gaëlle, a graphical designer, was asked to draw a three-tab system dedicated to a specific interactive application. She has started to freely explore an "aqua-like" style and wants to study different resizing behaviors for her graphics. She plans to use IntuiKit's Artistic Resizing Viewer, a small aplication that can load any non-interactive SVG graphics then allow to dynamically resize it with a handle.

When Gaëlle checks how her single graphical object is resized by IntuiKit, she first finds that compressing text is not visually acceptable (Figure 5 on the left). So she decides to explore horizontal resizing as a first step.

In her drawing program, she copies and pastes her graphics and decides to try the effect of simply compressing the tabs without resizing them (Figure 5, right pencil). She ungroups the graphics and moves the middle and right tab to the left so that they partially cover each other. She then tests the transition between the specified examples, as well as the effects of extrapolation (right part of Figure 5). She notices tabs are separated on one side and the text is hidden on the other side.

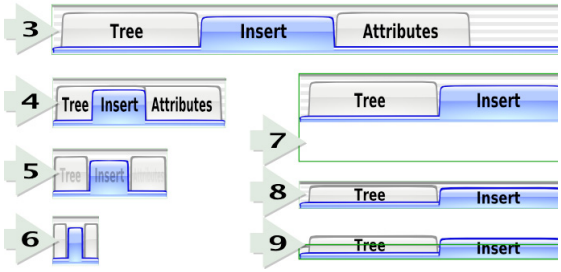

Figure 6: Scenario, continued: examples 3, 4, 5 and 6 describe the behavior for smaller and larger widths, whereas 7, 8 and 9 specify the vertical behavior.

The next iterations are shown on Figure 6: **3** Gaëlle does not want the tabs to expand when enlarged, so starting from the first example then expands the background and the bottom line to the right while keeping the tabs intact. **4** Gaëlle also wants to prevent the text from being hidden. She adds a smaller example in which the middle tab has been shrinked and the captions slightly compressed.

The resizing behavior is now as follows: when the tab system is enlarged, the left part remained unchanged and the background fills the remaining space. As the tab system gets compressed, the tabs begin to move behind each other at some time. Then, the middle tab shrinks as if labels were pushing it on both sides. The text continues to shrink but not quickly enough and eventually extend beyond the tabs.

The final additions are (Figure 6): **5** At smaller sizes, tabs become individual buttons with no text. This illustrates how subparts can be slowly or instantly hidden by controlling opacity. **6** is a "stabilizing" example. Such examples are added at extremal sizes to avoid unexpected extrapolation effects such as overshooting. A stabilizing example can be a copy of a previous example with a simple scale applied, or an exact copy with a different bounding box (in which case the object stops shrinking or growing).

7, **8** and **9** describe a simple horizontal resizing behavior: the tab system is not growing horizontally and stays aligned with the top (this is also specified by changing the bounding box alone). Tabs can shrink a little but are quickly pushed outside the bounding box. Though this behavior looks complex, we saw that it only needed nine examples and took less time to produce than the graphics alone[1].

At this point, Gaëlle can refine the Artistic Resizing she built or test completely new ones. She can also start realizing the final SVG file with the alternative graphical states she will send to Paul, the programmer, knowing that she can easily reuse graphics as well as existing Artistic Resizing specifications she made.

Gaëlle can also decide to explore a brand new graphical style, but in this case she will probably have to rebuild a resizing specification from scratch. Similary, she is able to iterate on the aqua-style graphics by reflecting changes in all examples, but it can be a tedious task if the number of examples is large. This is why the Artistic Resizing design process is best split in two stages: a first stage for designing the graphics in a given size and a second one for specifying how it will look like at other sizes.

A SIMPLE GEOMETRY INFERENCE SYSTEM

In this section, we describe the technique we used for inferring geometries by using bounding boxes as input variables. We explain exactly what the algorithm does, what it can and cannot infer, and why.

Extracting Affine Transformations

In the world of vector graphics, geometry is most often described by *affine transformations*. Example of such transformations are translation and scale, useful for coding location and size of graphical primitives, as well as rotation and shear. Affine transformations are most often formulated using homogeneous coordinates so that the six coefficients that characterize the transformation can be bundled into a single *transformation matrix* T:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} a^{11} & a^{12} & a^{13} \\ a^{21} & a^{22} & a^{23} \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{T}} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Some graphical authoring applications like Sodipodi maintain a transformation matrix for each primitive or group, while others like Adobe Illustrator maintain flattened lists of coordinates [12, 2]. In the last case, each isolated point can be seen as the origin point with a specific translation matrix. Therefore, each time a graphic designer is moving or scaling objects in any drawing application, he is manipulating affine transformation matrixes. More precisely, he is sequentially manipulating fixed sets of matrix coefficients, depending on the type of transformation chosen among available tools, interactors and keyboard modifiers.

In this section we suppose that during the Artistic Resizing design process, graphic designers are using their authoring application as a matrix manipulation tool. Everything else, including structure and graphical attributes, are kept constant from one example to another one (see Figure 7). This is easily done by the exclusive use of copy and paste and a subset of the available manipulation tools.

As a consequence, a graphical example can be viewed as a mere set of transformations matrixes and inferring examples simply requires interpolating affine transformations. Because examples share the same structure, related transformations such T_1 and T_1' in Figure 7 can be easily extracted, compared with each other and if they differ, independently interpolated to infer local varying laws.

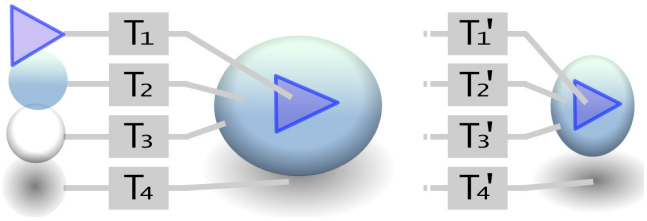

Figure 7: Two graphical examples that only differ by the affine transformations applied to their subparts.

Interpolating Affine Transformations

Suppose that a transformation T is a function of a scalar t representing, e.g., time. We call t the *input variable* and T the *output variable*. Now suppose we are given two points (t_1, T_1) and (t_2, T_2). Interpolation techniques allow us to infer the matrix T for each t between t_1 and t_2, the most simple of those techniques being linear interpolation[2] of the matrix coefficients [25]. Such techniques are easily generalizable to more than two examples, e.g., using piecewise linear interpolation [7].

Artistic Resizing involves *two* input variables: a width and an height, and we need to infer the transformation associated to any $(width, height)$ pair. Most monovariate interpolation methods cannot be used with more than two examples because there is no total order on the 2D space. As an example, piecewise linear interpolation cannot be applied without the help of triangulation techniques [16].

However, we make a simplifying hypothesis that brings us back to the case of two monovariate linear interpolations. This hypothesis is that *width* only impacts the three coefficients of the transformation matrix that contribute to the *x-coordinate* of the transformed point (the first line of the ma-

[1] Videos showing the examples described in this paper are available at http://www.intuilab.com/artresize.

[2] For other values of t, this operation is usually called extrapolation. Though we will be talking about interpolations, the problem is the same.

trix, see previous section). Similarly, the three coefficients on the second line are linearly interpolated along height. We call this simple transformation interpolation technique *orthogonal interpolation*.

The Orthogonal Interpolant

Together with a mathematical definition of orthogonal interpolation, we introduce here the notion of compatibility between examples in terms of orthogonal interpolation.

Definition 1. We define an *example* as a tuple $E = (w, h, T)$, where $w \in \mathbb{R}, h \in \mathbb{R}$ and T is a 2-D transformation matrix.

Definition 2. Two examples $E_1 = (w_1, h_1, T_1)$ and $E_2 = (w_2, h_2, T_2)$ are said *compatible* in terms of orthogonal interpolation iif

$$\begin{aligned} w_1 = w_2 \;\Rightarrow\; \{a_1^{1j} = a_2^{1j}\}_{j\in[1,3]} \\ \text{and} \quad h_1 = h_2 \;\Rightarrow\; \{a_1^{2j} = a_2^{2j}\}_{j\in[1,3]} \end{aligned}$$

where a_1^{ij} and a_2^{ij} are the coefficients of the matrices T_1 and T_2 respectively.

Definition 3. Let $E_1 = (w_1, h_1, T_1)$ and $E_2 = (w_2, h_2, T_2)$ be two compatible examples. The *orthogonal interpolant* of E_1 and E_2 is the function:

$$I^{orth}_{E_1,E_2} : (w, h) \;\mapsto\; T =$$

$$\begin{bmatrix} I^{lin}_{(w_1,a_1^{11}),(w_2,a_2^{11})}(w) & I^{lin}_{(w_1,a_1^{12}),(w_2,a_2^{12})}(w) & I^{lin}_{(w_1,a_1^{13}),(w_2,a_2^{13})}(w) \\ I^{lin}_{(h_1,a_1^{21}),(h_2,a_2^{21})}(h) & I^{lin}_{(h_1,a_1^{22}),(h_2,a_2^{22})}(h) & I^{lin}_{(h_1,a_1^{23}),(h_2,a_2^{23})}(h) \\ 0 & 0 & 1 \end{bmatrix}$$

where:

- $I^{lin}_{(x_1,y_1),(x_2,y_2)}$ is the linear interpolant of (x_1, y_1) and (x_2, y_2), i.e., the function $(x, y) \mapsto \frac{(x_2-x)y_1+(x-x_1)y_2}{x_2-x_1}$ if $x_1 \neq x_2$ and $(x, y) \mapsto y_1$ otherwise.
- a_1^{ij} and a_2^{ij} are the coefficients of the matrices T_1 and T_2 respectively.

The definition can be immediately generalized to more than two examples (i.e., extended to $I^{orth}_{E_1,\ldots,E_n}$) by replacing I^{lin} by the piecewise linear interpolant.

Geometrical Interpretation

Whereas orthogonal interpolation operates on affine transformations, it is useful to reintroduce concrete geometrical objects (i.e., points, lines, shapes) for understanding its effects on graphics. In this section, we give a geometrical interpretation of the previous definition.

Let T_1 and T_2 be two affine transformation matrices and $I_{T_1,T_2} = I^{orth}_{(w_1,h_1,T_1),(w_2,h_2,T_2)}$ one orthogonal interpolant that generates a family of intermediate transformations (there are as many interpolants as values of w_1, h_1, w_2, h_2).

If P_1, P_2 and P are the points obtained by transforming the *same* point of the plane through T_1, T_2 and the intermediate transformation $I_{T_1,T_2}(w, h)$ respectively, then $P(x, y)$ can be directly built from $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$ using the following formula:

$$P\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} I^{lin}_{(w_1,x_1),(w_2,x_2)}(w) \\ \\ I^{lin}_{(h_1,y_1),(h_2,y_2)}(h) \end{pmatrix}$$

That is, P simply results from a double linear interpolation of P_1 and P_2 along the x-axis and the y-axis separately, as illustrated on Figure 8.

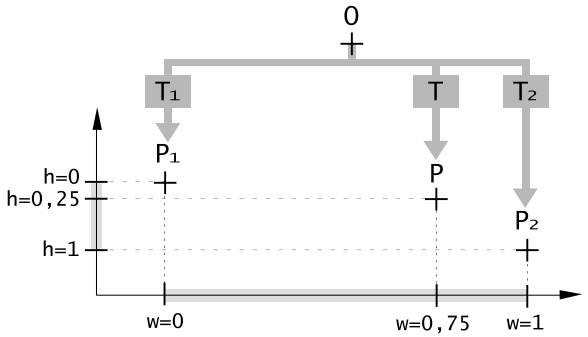

Figure 8: Geometrical interpretation of orthogonal interpolation. Here, an intermediate transformation T is applied to a point O, with $T = I^{orth}_{((0,0),T_1),((1,1),T_2)}(0.75, 0.25)$.

This stands true when intermediate transformations are applied to shapes or tuples of points: computing an interpolated transformation I_{T_1,T_2} before applying it to a tuple of points is the same as applying the two example transformations T_1 and T_2 then interpolating each point as previously described.

Graphics with Multiple Transformations

What is the result of orthogonal interpolation when applied on graphics made of several shapes, each having its own transformation? Let us model shapes as tuples of points and consider two tuples of shapes $(S_1^1,\ldots,S_1^n)$ and $(S_2^1,\ldots,S_2^n)$ obtained after applying $(T_1^1,\ldots,T_1^n)$ and $(T_2^1,\ldots,T_2^n)$ to the original tuple of shapes.

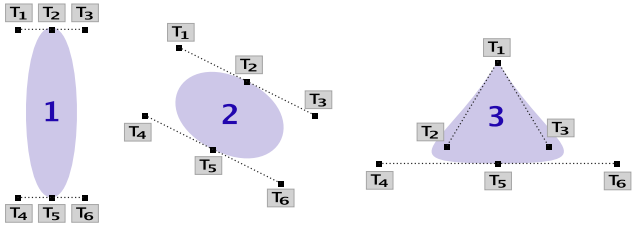

Figure 9: A spline morphing with three key images, using a transformation on each control point. Interpolating control points is the same as interpolating the curve.

Orthogonal interpolation is easily generalized to tuples of transformations by interpolating each pair (T_1^i, T_2^i) separately. Again, intermediate composite objects can be built by direct interpolation of each pair of points (P_1^i, P_2^i), with $(P_1^1,\ldots,P_1^m)$ and $(P_2^1,\ldots,P_2^m)$ being all the points obtained by flattening $(S_1^1,\ldots,S_1^n)$ and $(S_2^1,\ldots,S_2^n)$.

One important consequence of this is that orthogonal interpolation is independent from the way the points (or transformations) are partitioned into shapes. Interpolating graphics from Illustrator or Sodipodi would give the same visual effects. This is illustrated in Figure 9: orthogonal interpolation between 1 and 2 involves multiple transformations but would give the same results if a single transformation were associated to the set of control points (or even to the whole curve, due to the property of affine invariance of spline control points). However, one transformation is not enough for the interpolation between 2 and 3.

Note, however, that there is no more equivalence when orthogonal interpolation is further generalized to compositions of transformations. Moreover, the resulting interpolants may not be linear any more: for example, if an object is scaled by $1/2$ then 2 on the first example and scaled by 2 then $1/2$ on the second one, the actual scale on both examples is 1 while being $3/2 \times 3/2$ inbetween. This is not acceptable because such behavior is unlikely to be expected and most interesting properties are lost. Fortunately, this problem can always be addressed by pre-multiplying all the varying matrices down the transformation trees before interpolating them. Provided that scene graphs are normalized in this way, interpolation results are fully independent from the way the graphical object has been structured into groups, shapes, or coordinate lists.

Useful Properties of Orthogonal Interpolation

What if a graphic designer draws a square centered into another square as a first example and keeps it centered in the second example ? Will the square remain centered on all interpolated images, as he may expect ?

A *conservative property* for orthogonal interpolation is a geometrical property that, if verified on two tuples of shapes obtained after applying $(T_1^1,...,T_1^n)$ and $(T_2^1,...,T_2^n)$, is still verified after applying any of the intermediate tuples of transformations built from $(T_1^1,...,T_1^n)$ and $(T_2^1,...,T_2^n)$. Interesting conservative properties can be deduced from the "partition independence" property and the fact that orthogonal interpolation preserves affine combinations of points. Some of those properties are (see Figure 10):

- *a. Preservation of projected algebraic measures.* If once projected on a given axis, a vector has the same algebraic measure on the two examples, then this value will remain constant. As a result, "horizontal" and "vertical distances" are preserved, provided that the points are kept on the same side from each other. This allows specification of borders and margins.
- *b. Preservation of relative ratios.* If three points are aligned by the same distance ratio on the two examples, then they will remain aligned and the distance ratio will remain the same. As a consequence, midpoints are preserved. This property is useful as constant ratios are likely to be used by graphic designers.
- *c. Preservation of coincident vertices.* If two points coincide on the two examples, then they will always coincide. As a consequence, two shapes which meet through the same pair of points on each example will remain in contact. This allows specifying incidence relationships between different graphical primitives.
- *d. Preservation of parallelism.* If two lines are parallel on the two examples, then they will always be parallel. Designers may particularly expect this invariant for horizontal and vertical lines, even if it stands true for any orientation. Parallelograms are also preserved, as well as rectangles parallel to the main axes.
- *e. Preservation of affine combinations.* This general property is unlikely to be exploited as such by the graphic designer but has been added to Figure for illustrative purposes. Please note that alignment is a conservative property only if affine combination is respected.

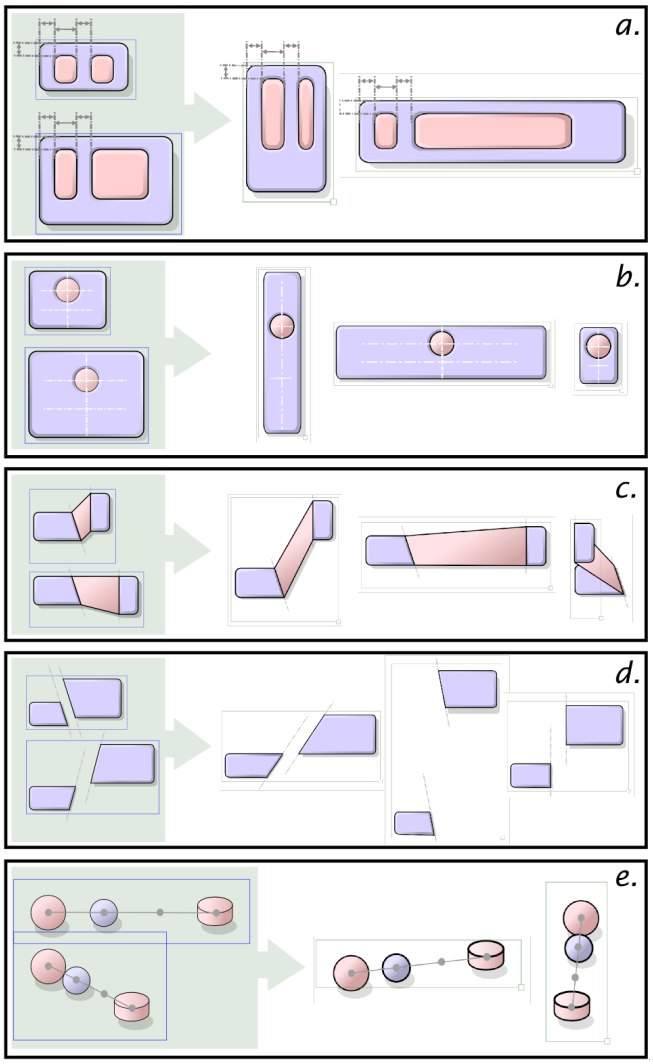

Figure 10: Conservative properties illustrated: *a.* preservation of projected algebraic measures, *b.* preservation of ratios, *c.* preservation of coincident vertices, *d.* preservation of parallelism, *e.* preservation of affine combinations.

Limitations of Orthogonal Interpolation

The main limitations of orthogonal interpolation in terms of conservative properties and expressive power are:

- *a. Composition.* As seen before, orthogonal interpolation poorly handles composite transformations. Nonlinearity arises if the transformation law of an object is

spread among its different ancestors, which typically happens when examples are individually grouped, ungrouped and manipulated several times. This limitation is easily overcome by pre-multiplying transformations in the scene graph, but it can be an issue if local transformations need to be kept in the GUI for some reason.

- *b. Axis Dependency.* Orthogonal interpolation is dependent from the coordinates system: some properties, such as preservation of right angles, are only true in a given orientation. As a consequence, some geometrical invariants may disappear if graphics are rotated.
- *c. Singularities.* Intermediate matrices can (at least theoretically) become singular: if two examples contain an object and its reflection, the object exactly halfway inbetween will be infinitely thin. It might be a problem when inverting matrices for object picking, for instance.
- *d. Rotations.* Orthogonal interpolation poorly handles rotations: an object cannot be told to simply rotate from one angle to another one. Instead, the two shear components are interpolated separatedly which causes the object to be deformed.
- *e. Cross-Axis constraints.* Orthogonal interpolation cannot infer constraints involving both axis. For example, it is not possible to specify constant ratios such as squares and circles, or oblique lines with a constant direction. Similarly, an object cannot be told to expand in the orthogonal direction when compressed, which may be needed for certain squeezing effects.
- *f. Non-linear constraints.* Orthogonal interpolation is not able to infer behaviors allowed by non-linear layout models such as flow layout. For example, it is not possible to tell an object to "carriage return".

Limitations *b*, *c* and particularly *d* have been pointed out previously by [25], which compared a monovariate technique close from orthogonal interpolation with a technique involving separate interpolation of rotation. However, this study has been done in the context of animation (cartoons and 3D animation, not GUIs), which involves quite different issues:

- Animation is monovariate (time) whereas resizing is bivariate (width and height).
- GUI objects have strong horizontal and vertical components, they are "box-like". Bounding boxes used in resizing interactions are themselves a strong reference system with a fixed orientation. Among other consequences, graphic components rarely rotate (at least in a continuous way) when the object is resized.
- In cartoons and 3D animation, emphasis is put on rigidity, that is why rotation is preferred to shear. In GUIs, objects have a "elastic" feel, they are expected to be deformable. Shear may be desirable to preserve common vertices, a property polar interpolation lacks.
- Animations can be precomputed, whereas in most cases we want resizing to update interactively. Separate interpolation of rotation involves polar decomposition, which is more costly than orthogonal interpolation.

Finally, we argue that limitations *e* and *f* (lack of expressive power) are the price to pay for simplicicy. The Chimera [13] system we already mentioned has a more sophisticated inference system and is able to infer cross-axis and rotational constraints, but it needs more examples for solving ambiguities. In constrast, orthogonal interpolation needs only two or three examples for inferring resizing behaviors most commonly seen in GUI widgets today. Moreover, several examples can be used in Artistic Resizing for describing non-linear (piecewise linear) resizing behaviors whereas all examples in Chimera are used to infer a single set of constraints. In fact, when compared to by-example systems our technical approach is quite new: instead of explicitly searching for invariants, we use a simple interpolation technique that preserves invariants as a natural consequence of its mathematical properties. One of the advantages is that both inference and resizing computations are very fast, even with extremely complex vector graphics.

GENERALIZATION AND FUTURE WORK

Though Artistic Resizing has not been designed to express all layout management mechanisms, we describe in this section how it can be combined with more classical layout management systems. We also consider possible approaches for interpolating vector graphics more extensively, by taking other graphical attributes into account. Finally, we describe other future work.

Combination with Other Layout Models

As a proof of concept, we extended Artistic Resizing so as to support containement hierarchies and collaborate with a layout manager that distributes available space among children. In Artistic Resizable containers, the area allowed for children is depicted by a rectangle whose variation law is specified visually as any other graphical subpart. Changes in this rectangle are propagated to a layout manager which in turn updates the sizes of its Artistic Resizing children.

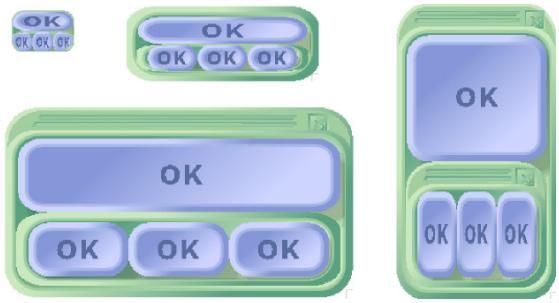

Figure 11: Nested Artistic Resizing.

Figure 11 shows an example of nested panels and buttons. The panel has been graphically designed so that the title bar slides behind the panel below a given height. Borders also shrink at very small sizes to give full room to children. All behaviors smoothly blend with each other to form a whole object with sophisticated resizing. Figure 12 illustrates another example of cooperation between Artistic Resizing and an AquaDock-like layout manager: every time the mouse pointer moves on the dock, the layout updates the size of each icon.

For now, Artistic Resizing only supports top-down (or outside-in) layout propagation. One example of bottom-up (or inside-out) layout strategy is when a button expands horizontally when its label is too wide [9]. We are currently in-

Figure 12: An "Artistic Dock".

vestigating the integration of such mechanism, together with bidirectional layout propagation involving negociation [22].

Extensive Interpolation of Scalable Vector Graphics

In the previous section we described the main inference mechanism in Artistic Resizing, i.e., interpolation of affine transformations. We applied this technique to SVG files by interpolating all the SVG attributes associated to the X and Y axes (widths, heights, coordinates of curve control points). We additionnaly identified the following attributes that would need to be managed so as to be able to fully interpolate arbitrary SVG pictures:

- *Other size-related features:* some features such as circle radii or stroke widths are spatial dimensions but are not related to the X and Y axes. Consequently, the hypotheses that underly orthogonal interpolation do not apply. A simple approach to handle that would be to combine the width and height into a single variable that reflects the size of the bounding box, so as to use it as input for piecewise linear interpolation. For example, Artistic Resizing currently interpolates stroke widths using the bounding box area as the input variable.
- *Non-spatial features:* the only non-spatial feature handled by Artistic Resizing is opacity, which can be interpreted as related to size (the smaller the object, the more transparent it is, for instance). Other features such as colors or text content have no spatial semantics, so orthogonal interpolation does not apply either. Possible solutions are discussed later in this section.
- *Structural features:* for now Artistic Resizing does not interpolate SVG structures: subparts cannot be suppressed from or added to individual examples. A simple extension would be to allow alternative graphical representations by separately interpolating examples with different structures. Another more sophisticated approach would be to perform pattern matching on SVG subtrees.

In the search for a more general solution, generic multivariate interpolation methods may appear interesting, particularly for interpolating features that are not directly related to X or Y axes. But they also have drawbacks.

Bilinear interpolation, for example, would require that all $(width, height)$ pairs fill a regular grid. For arbitrary distributions of sample points, the most popular methods are Delaunay triangulation with linear interpolation and Natural Neighbor interpolation [16]. They are efficient for smoothing big data sets (e.g., a hundred of points) but lack stability and predictability when the number of examples is very small (e.g., between two and ten). As a consequence, they may be difficult to predict and to manipulate by graphic designers.

Polynomial and spline-based interpolants also deserve to be explored for enhancing smoothness in general, but when applied to orthogonal interpolation most conservative properties are lost because of non-linearity.

Other Future Work

So far, we have mainly applied the Artistic Resizing approach on dedicated, static user interface objects. For instance, the programmer cannot change the captions nor add more tabs on the object of the sample scenario. Future work includes extensive support for parametrization in order to provide a set of reusable Artistic Resizable widgets.

Artistic Resizing is still young and experimental and needs to be tested extensively with graphic designers. We are first planning to reduce the experimentation-evaluation loop by allowing designers to operate directly on the interpolated examples. This can be done by extending an authoring application with a plug-in for Artistic Resizing. In this paper, we mainly focused on traditional resizing behaviors relying on visual invariants and explained how the technique supports them. We nonetheless suspect that as a new creativity tool, Artistic Resizing can lead to much more original interfaces once put in the hands of graphic designers.

CONCLUSION

We have described artistic resizing, a visual programming-by-example technique that allows graphic designers to describe the pixel-precise, non linear resizing of visual objects by using their familiar drawing tool. Their work is interpreted and transformed into an executable form by an inference and interpolation algorithm. The inference and interpolation algorithm is simple and predictable and preserves a number of useful properties of the provided examples. Incorporated in the model-driven approach proposed by the IntuiKit environment, artistic resizing gives designers greater control over the graphical part of applications. It further increases their role as “programmers” in multidisciplinary software engineering groups.

ACKNOWLEDGEMENTS

This article was helped by discussions with Yves Rinato (Intactile Design), as well as work with Stéphane Sire, Pham Nguyen Khang and Bruno Merlin. Many thanks to Michel Beaudouin-Lafon, Stéphane Conversy, Sandra Basnyat and Frédéric Jourdan for their comments on this paper and to Céline Schlienger and Alexandre Lemort for helping in the production of the examples.

REFERENCES

1. I. Adobe Systems. Designing multiple master typefaces. http://www.adobe.com/.

2. I. Adobe Systems. Adobe Illustrator CS scripting guide. http://partners.adobe.com/public/developer/en/illustrator/sdk/IllustratorScriptingGuide.pdf, 2003.

3. G. J. Badros, J. J. Tirtowidjojo, K. Marriott, B. Meyer, W. Portnoy, and A. Borning. A constraint extension to scalable vector graphics. In *World Wide Web’10*, pages 489–498, May 2001.

4. T. Beier and S. Neely. Feature-based image metamorphosis. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 35–42, New York, NY, USA, 1992. ACM Press.

5. L. Cardelli. Building user interfaces by direct manipulation. In *UIST '88: Proceedings of the 1st annual ACM SIGGRAPH symposium on User Interface Software*, pages 152–166, New York, NY, USA, 1988. ACM Press.

6. S. Chatty, S. Sire, J. Vinot, P. Lecoanet, C. Mertz, and A. Lemort. Revisiting visual interface programming: Creating GUI tools for designers and programmers. In *Proceedings of the ACM UIST*, pages xxx–yyy. Addison-Wesley, Oct. 2004.

7. J. W. Harris and H. Stocker. *The Handbook of Mathematics and Computational Science*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

8. S. E. Hudson and C.-N. Hsi. A synergistic approach to specifying simple number independent layouts by example. In *CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 285–292, New York, NY, USA, 1993. ACM Press.

9. S. E. Hudson and S. P. Mohamed. Interactive specification of flexible user interface displays. *ACM Transactions on Information Systems*, 8(3):269–288, 1990.

10. S. E. Hudson and I. Smith. Ultra-lightweight constraints. In *Proceedings of the ACM UIST*, pages 147–155, New York, NY, USA, 1996. ACM Press.

11. S. E. Hudson and K. Tanaka. Providing visually rich resizable images for user interface components. In *Proceedings of the ACM UIST*, pages 227–235, 2000.

12. L. Kaplinski, F. Felfe, M. Oka, and M. Yamato. Sodipodi. http://www.sodipodi.com/, 2005.

13. D. Kurlander and S. Feiner. Inferring constraints from multiple snapshots. *ACM Transactions on Graphics*, 12(4):277–304, 1993.

14. M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, pages 8–22, Feb. 1989.

15. MacroMedia Press. *MacroMedia Director 6 and lingo authorized*. Addison-Wesley, 1997.

16. E. Meijering. A chronology of interpolation: From ancient astronomy to modern signal and image processing. *Proceedings of the IEEE*, 90(3):319–342, March 2002.

17. K. Miyashita, S. Matsuoka, S. Takahashi, A. Yonezawa, and T. Kamada. Declarative programming of graphical interfaces by visual examples. In *Proceedings of the ACM UIST*, pages 107–116. ACM Press, 1992.

18. B. A. Myers. Creating user interfaces using programming by example, visual programming, and constraints. *ACM Trans. Program. Lang. Syst.*, 12(2):143–177, 1990.

19. B. A. Myers, J. Goldstein, and M. A. Goldberg. Creating charts by demonstration. In *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 106–111, New York, NY, USA, 1994. ACM Press.

20. B. A. Myers, R. G. McDaniel, R. C. Miller, A. S. Ferrency, A. Faulring, B. D. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The Amulet environment: New models for effective user interface software development. *IEEE Transaction on Software Engineering*, 23(6):347–365, June 1997.

21. T. Ngo, D. Cutrell, J. Dana, B. Donald, L. Loeb, and S. Zhu. Accessible animation and customizable graphics via simplicial configuration modeling. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 403–410, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

22. D. R. Olsen. *Developing User Interfaces.* Morgan Kaufmann Publishers, 1998.

23. C. Rose, M. F. Cohen, and B. Bodenheimer. Verbs and adverbs: Multidimensional motion interpolation. *IEEE Comput. Graph. Appl.*, 18(5):32–40, 1998.

24. T. W. Sederberg and E. Greenwood. A physically based approach to 2d shape blending. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 25–34, New York, NY, USA, 1992. ACM Press.

25. K. Shoemake and T. Duff. Matrix animation and polar decomposition. In *Proceedings of the conference on Graphics interface '92*, pages 258–264, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

26. R. Williams. *The Animator's Survival Kit: A Manual of Methods, Principles, and Formulas for Classical, Computer, Games, Stop Motion, and Internet Animators*. Faber & Faber, 2002.

27. G. Wolberg. Image morphing: a survey. *The Visual Computer*, 14(8/9):360–372, 1998.