Figure 2.3: *Example strokes and a completed illustration with simulated brushes. Figure from Strassmann [1986].*

Figure 2.4: *An example sculpture presented in Galyean and Hughes' 1991 paper.*

2.2 Life Imitates Art

I now shift my description to a different aspect of computer graphics tools.

Again, we begin with Sketchpad. While Sketchpad was revolutionary for being the first computer-aided drawing tool, it was certainly not the first drawing tool.[4] Sketchpad, like so many tools after it, was designed – in part – by copying a process (drafting) from the real world and adding features here and there (instancing, constraints, undo).

When digital painting emerged in the early 1970s, systems relied on a primitive imitation of real-world painting (with static-shaped, solid-colored brushes) [Shoup 2001]. This changed with the introduction of natural media paint programs, which seek to imitate more exactly real art materials. The first[5] of these was the Hairy Brushes work of Strassmann [1986], in which a sumi-e brush is simulated as a collection of strands (Figure 2.3). These days, commercial programs exist which replicate real-world painting effects (e.g., [Fractal Design 1992]), and researchers have continued to emulate and perfect the more subtle phenomena of real-world painting (e.g., watercolor diffusion [Curtis et al. 1997]).

[4] I expect that humor probably belongs to a muddled hand or chalky rock.

[5] An earlier work allows one to virtually paint with real objects, using a prism and video camera [Greene 1985].

Figure 2.2: *A DEC GT-40 terminal, introduced in 1972, features many of the same I/O capabilities as Sutherland's Sketchpad system including vector-based graphics and a light pen input device. Image from the 1976 PDP-11 Peripherals Handbook, as appears at http://www.brouhaha.com/~eric/retrocomputing/dec/gt40/handbook.html .*

The framebuffer was an important innovation because it provided a (conceptually, at least) uniform access to graphics output hardware: a grid of pixels. Software running on a computer equipped with a framebuffer could display any image at all (up to the resolution of its attached display); all that remained was to determine how to fill the buffer with meaningful data.

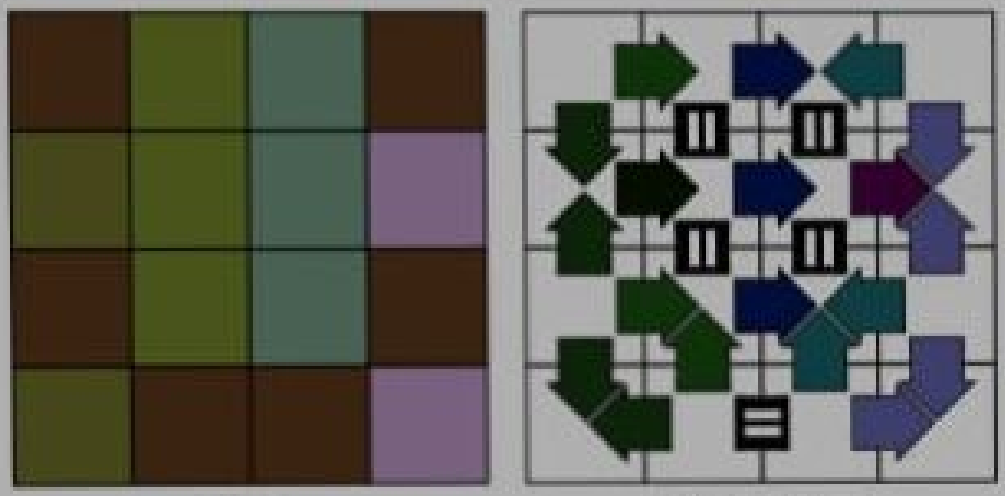
A grid of pixels leaves a lot of open space to innovate in drawing primitives (because, for many tasks, a grid of pixels isn't the ideal representation) and hardware acceleration (because, well, there are a lot of pixels in the framebuffer and CPUs are only so fast). One finds a snapshot of these two threads in the proceedings from the inaugural SIGGRAPH conference in 1974: a proposed set of common subroutines for graphics programs [Smith 1974], a high-level description for interaction and display of 3D objects [Staudhammer and Ogden 1974], and a proposal for a graphics processor [Eastman and Wooten 1974] are all in evidence.

By the mid-80s, graphical framebuffer-style displays were the display standard not just for research prototypes but also for the emerging personal computer market as well.[1] This is unsurprising, since, with cheap memory, a framebuffer is a very cost-effective display system – both because few separate components are required to generate a display signal and because, with the proper design, end-users could plug their computer into a television instead of buying a new display.

IBM's series of graphics adaptors (1981: Color Graphics Adaptor; 1984: Enhanced Graphics Adaptor; 1987: Video Graphics Array) provided the interface standard for PC framebuffers. Software written to the CGA,

[1] I elide, herein, a discussion of text- or tile-based displays where a framebuffer-like rectangular array indexes blocks of colors instead of simply colors. Such displays pre-date framebuffers and essentially trade away flexibility for lower memory overhead.

1.1 Painting with Edges



colors color differences

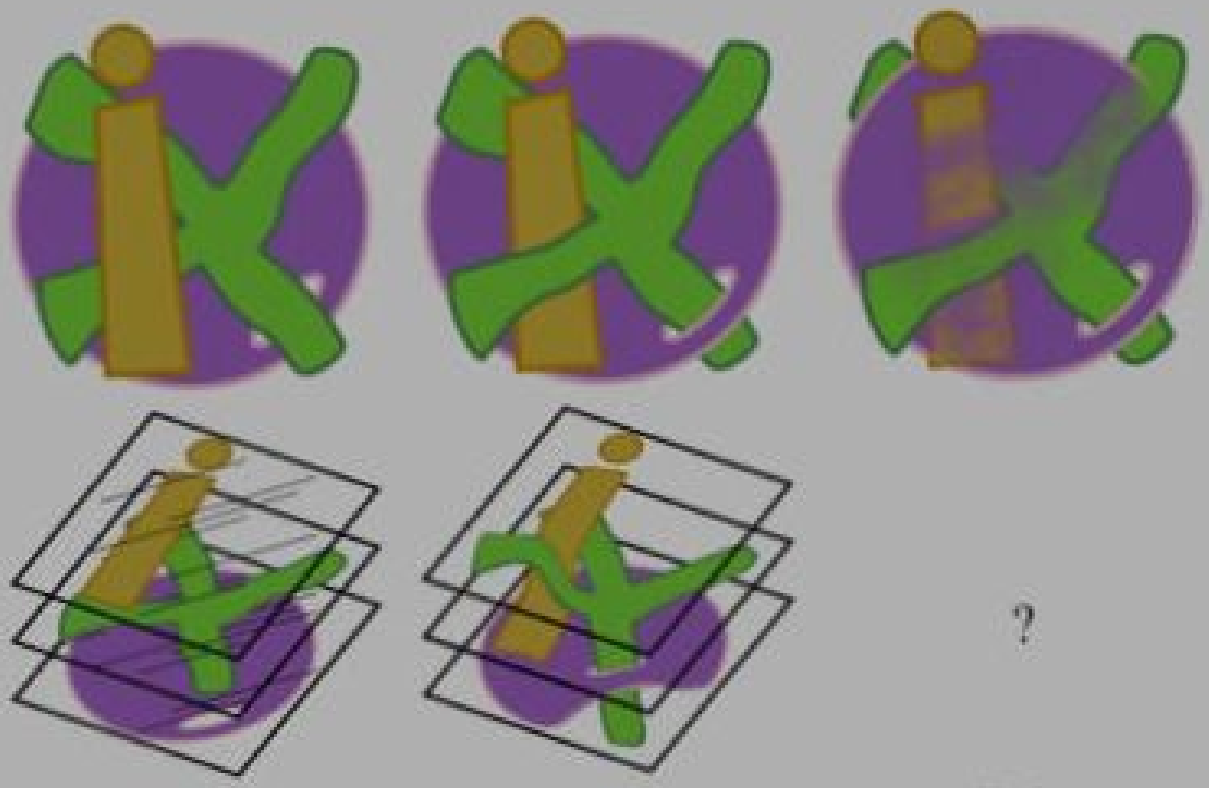
Humans have been editing the local color values of surfaces for over six millennia, from early cave drawings to modern paintings. And so when primitive graphics researchers began to develop paint systems, it seemed natural to allow the same sort of interaction – virtual brushes that apply user-selected colors to a framebuffer canvas. Modern paint systems such as the GIMP still operate in this way; natural media paint systems go even further by simulating the interactions of charcoal, pencil, and brush with the canvas.

The trouble is that all of the paint systems upon which digital painting is based – from pigment on cave wall to oil on canvas to airbrush on hot-rod – are display-motivated. To produce the impression of a color, the stone, canvas, or fender must reflect certain wavelengths of light and absorb others. In the physical world, the only way we can do this is by altering surface properties – e.g., by applying pigments. But with computer paint systems, image output is no longer a direct consequence of user input; rather, the computer acts as a moderator, deciding what to display in response to brush strokes. This means that having brush strokes change local color in a computer paint system wasn't a requirement – it was a choice.

What if we do something different? Perceptual studies have shown that the human visual system is sensitive to edges – that is, to differences between colors [Werner 1935; Land and McCann 1971]. In fact, many works in image compositing and manipulation exploit this notion to good effect (see Section 3.2). I've taken this idea further by designing a paint system that works on the differences between pixel colors instead of the colors themselves.

The computer has to work harder to display these images (it must solve a least-squares problem to arrive at an intensity image close to the desired differences). However, the resulting paint program allows artists to directly introduce relevant features (edges) while leaving irrelevant information (absolute color) undefined.

1.2 Stacking Images Like Paper and Like Fog



like film like paper like fog

Film compositing is a method of combining together elements imaged onto film into a final scene. For each element, a "mask" is created that blacks out all but the portion of the element that should be visible in the final scene. The filmstrips for the mask and element are stacked and fed into an optical printer which exposes portions of a negative. Successive elements are exposed onto the same negative to create a final image.

Digital compositing, as exists in today's image and video editing programs, works much the same way – graphical elements are drawn in some stacking order to produce a final image. Perhaps the main innovation of digital compositing is the notion of intrinsic alpha – the idea that each pixel carries with it a transparency value (instead of transparency information being carried on a separate "mask" filmstrip). Intrinsic alpha means that graphical primitives can be thought of as having shapes other than just "the whole image" – that is, objects *don't exist* where they are fully transparent.

And yet, we haven't exploited this notion to enhance the way we perform compositing. If we think of graphical objects as being cut-outs – as things with holes where there is zero alpha – then why do we still stack them as if they are pieces of film?

Local Layering is one of a few works to address this problem (others have used a knot-theoretic approach and tried to infer structure from an unstructured representation – see Section 4.2). I provide users access to a stacking model based on the idea of graphical objects being cut-outs that can freely pass through each

Figure 2.1: *Ivan Sutherland's Sketchpad. Picture from Sutherland [1963].*

graphical primitives. In Sketchpad lay a glimmer of the way forward: *service programs* (library routines) to perform basic drawing operations (lines, arcs, text), callable from both Sketchpad and from other programs.

However, such service routines were useless to a large community without some uniformity in basic computer hardware. Over the next decade, the development and proliferation of microcomputers to industrial labs and universities – notably DEC's PDP-8 and PDP-11 series – meant that basic interactive computing was widely available and somewhat uniform. However, there was still very little agreement on what, exactly, a graphics interface should be. By 1972, DEC was producing a graphical terminal, the GT40 (Figure 2.2), that had many of the basic drawing and digitization capabilities included in Sketchpad. That is to say, it produced graphics by drawing text and lines.

In the same time frame, Richard Shoup and others at Xerox PARC were developing a paint system, Superpaint, with an entirely lower-level display technology [Shoup 2001]. Superpaint was built around a wire-wrapped framebuffer attached to a Data General Nova 800 minicomputer (a contemporary to the PDP-11, built by ex-DEC employees). A framebuffer is radically different from a vector display like the GT40. Where vector displays are perforce somewhat intelligent (they need to be able to control a laser or electron beam, sweeping it along lines, arcs, even text), framebuffers are dumb memory. A framebuffer contains a sequence of memory locations, each memory location containing the color for a pixel on, say, an NT television.

1.2.2 Contributions

- Local Layering:
  - A simple formulation of the problem of consistent local stacking based on a graph of lists.
  - Local re-stacking operators and proofs of their correctness and sufficiency.
  - Both the operators and formulation extend trivially to spatio-temporal volumes for animation (proofs do not depend on planarity).
  - A temporal coherence scheme for maintaining the stacking order when layers are moved or their contents edited. This scheme can also propagate information about visibility edits in depth-peeled 3D models (Figure 1.7).
- Soft Stacking:
  - A continuous version of layer stacking allowing interleaving of volume-like layers.
  - An objective function that allows optimal continuous stackings to be achieved.
  - A layer-constraint dialog that allows artists to quickly specify a sub-space of stackings.
  - A brush that locally restricts stacking orders to a subspace.

Figure 1.11: *My 3D-like texturing method works on deforming objects.*

1.3.2 Contributions

- A method for animating 3D-like surface texture motion in a 2D animation program, informed only by local shape descriptors.
- The method does not make any assumptions about camera setup, projection type, or global scene consistency.
- A gizmo for local shape description sufficient to recover local projection parameters.
- A closed-form 2D manifold correspondence algorithm that does not require embeddings of the source or target manifolds.

1.3 3D-like Texturing for 2D Drawing

Current packages for creating animations on a computer generally fall into one of two categories: 2D animation software generally apes the traditional animation process, with frames created by drawing directly in the image plane; 3D animation software mimics cinema, with the complexity of actors, lighting, and set design.

One of the benefits of 3D animation is that the tools lend themselves to data re-use across frames. For example, a 3D model of a plant does not need to be re-built when a camera pans around it, while a 2D drawing of a plant would need to be. However, because of the disciplined structure (consistent scenes and projections, rigid skeletons and transformations) that permits this re-use, 3D animations often fall short of 2D animations in their expressiveness.

In this work, I address the re-use of surface texture in 2D animations. Re-drawing surface texture is cumbersome, especially if it is complex; an automatic approach would save artists time and permit them to use more surface detail. However, for a computer to be able to transfer surface texture from frame to frame it needs to know something about the 3D surface shape – intuition that conventional 2D animation systems lack.

In my system, the artist supplies the computer with this surface shape information through the use of gizmos built on local surface properties (tangents) that humans seem to be good at reasoning about.[2] In contrast to a 3D animation system, however, my 3D-like texturing approach does not require a globally consistent 3D model (or, indeed, even surface shape information for patches without texture). In this way, irrelevant 3D detail can be left fully ambiguous.

1.3.1 Results Preview

The figures below show examples of 3D-like texture transfer, emphasizing the importance of local shape cues (Figure 1.10), and demonstrating that my method handles deforming objects with aplomb (Figure 1.11).

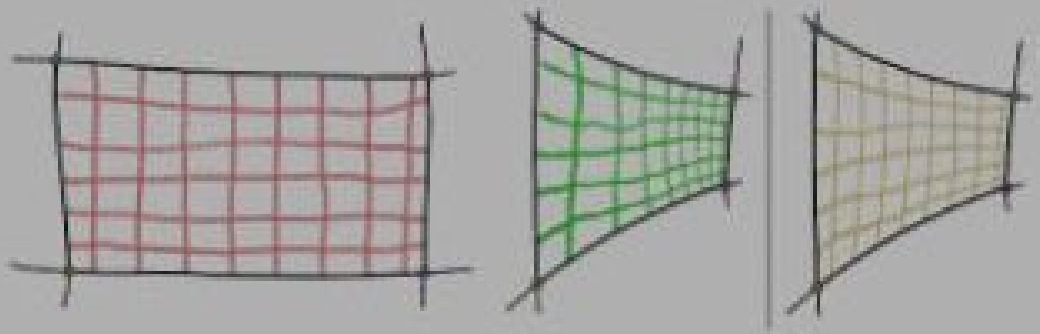


Figure 1.10: Left and middle, *Artist-specified shape cues allow for perspective-like effects in texture transfer. (Compare to,* **right**, *the same transfer with no shape cues.)*

[2] E.g., Koenderink [1992] shows that humans can estimate local surface normals repeatably, up to ambiguities due to light position. As tangents are related to normals in a straightforward way, one hopes this ease of estimation carries over.

other wherever they are fully transparent – that is, like figures cut from sheets of paper. This stacking is controlled by a modified layers dialog that changes the order of layers at a user-selected point in the image instead of globally. Thus, changing the ordering of elements in a given location is as simple as clicking on that location then re-ordering the elements in a familiar way.

Soft Stacking takes the concept further, allowing continuous rearrangements of layers (as if they were volumes of fog, and could appear partially in front of and partially behind other layers). I've devised two methods for manipulating these stackings – a brush-based approach which allows orders to be painted directly, and an optimization-based approach which solves for an optimal (collision-minimizing) ordering.

These systems both provide tools that specify stacking orders locally in a way that matches the artist's understanding of the world (paper can't pass through paper, "thick" fog passes through things more slowly than "thin" fog), and leave the global implications of these local edits for the computer to figure out.

1.2.1 Results Preview

The figures below show a local stacking (Figure 1.4), the local layers dialog (as compared to a standard layers dialog) (Figure 1.5), another stacking method where layers are dragged (Figure 1.6), an impossible 3D figure created by locally re-ordering a depth-peeled 3D model (Figure 1.7), a brush-based soft stacking result (Figure 1.8), and an optimization-based soft stacking result (Figure 1.9).

Figure 1.4: *A complex stack of four layers.*

1.1.1 Results Preview

The result images below show the gradient-domain paint brushes (Figure 1.1) available in Gradient Paint, and example sketches (Figure 1.2) and image edits (Figure 1.3) made with the system.

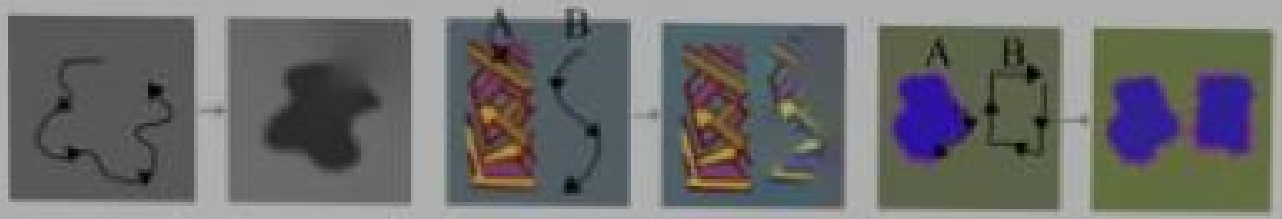


Figure 1.1: *The three paint brushes supported by Gradient Paint.* **Left:** *the gradient brush draws edges.* **Middle:** *the clone brush copies edges.* **Right:** *the edge brush copies and re-orients edges.*

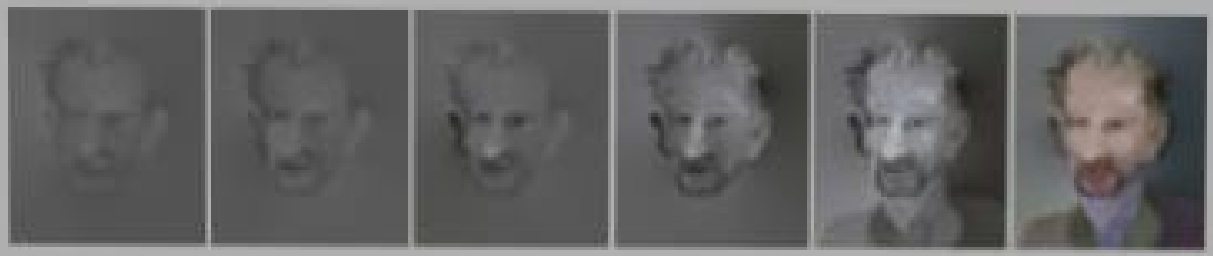


Figure 1.2: *Creating a new image by sketching with color differences. (Artist: M. Mahler.)*

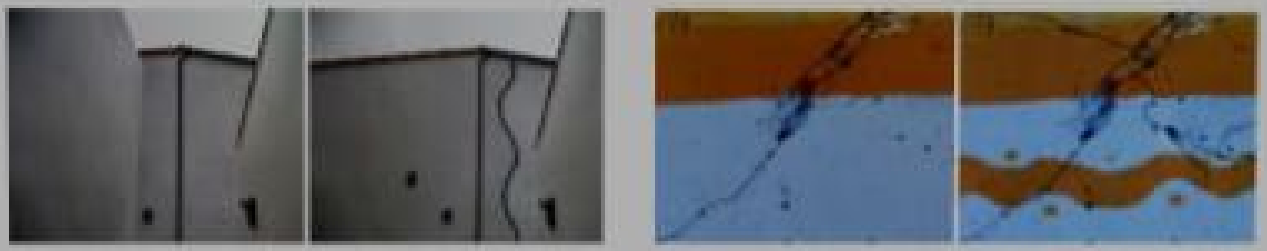


Figure 1.3: *Copying edges within an image to change a roofline and add more cracks. (Originals, leftmost in each pair, by clayjar (***left***) and Tal Bright (***right***) via flickr.)*

1.1.2 Contributions

- Proof-by-example that one can fit intensity images to gradient images fast enough to provide real-time feedback while editing (using a GPU-based multigrid, similar to that previously used in fluid solvers).
- A set of brushes for working in the gradient domain.

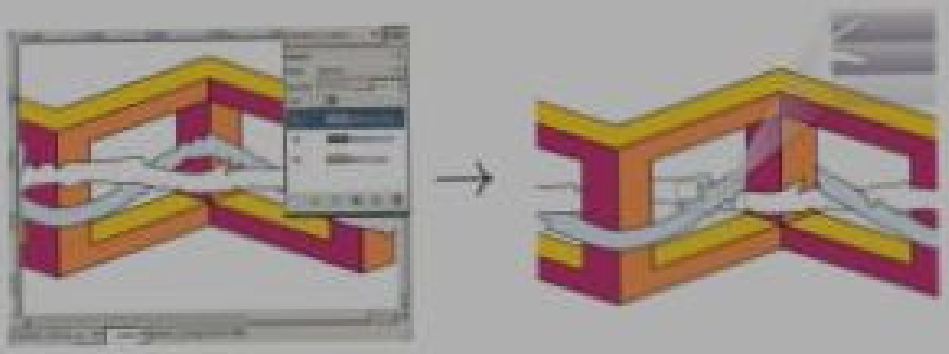


Figure 1.5: *I replace the global layers dialog with a local stacking dialog to allow local re-ordering.*

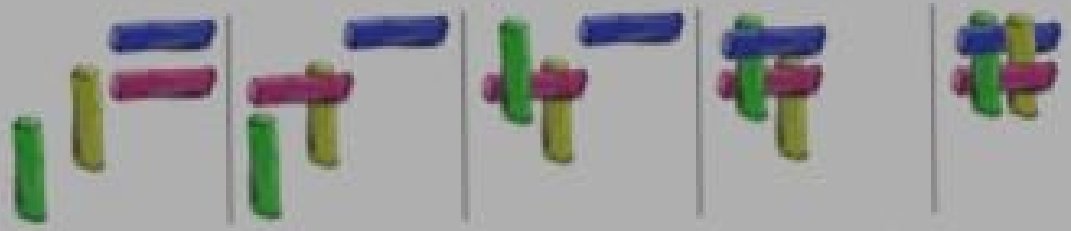


Figure 1.6: *Weaving layers by dragging them over each other.*

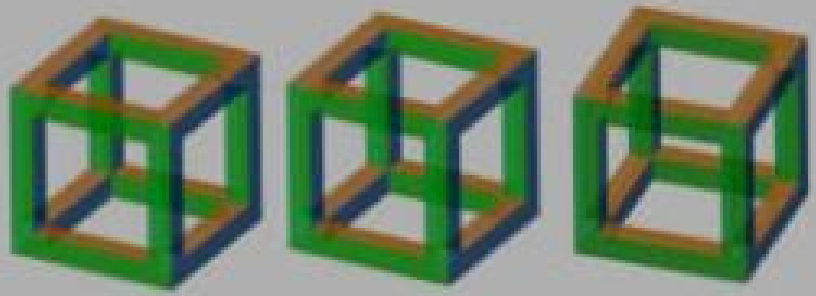


Figure 1.7: *An impossible figure created by locally changing depth order of a 3D model.*

Figure 1.8: *Soft Stacking allows partially-transparent things (like mist) to gently mix with other layers. The artist specified the interleaving of frog, snake, and apple by painting local stacking orders.*

Figure 1.9: *In this Soft Stacking result, the artist specified constraints at the intersections of the bolts and the system solved for an optimal global order. Note the smooth mixing of the halos around the bolts.*