Image Editing and Creation with Perception-Motivated Local Features

James Lewis McCann

July 2010
CMU-CS-10-130

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This research was sponsored by the National Science Foundation under grant numbers CCF-0702443 and IIS-0326322. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: ambiguity, animation, artist tools, gradient domain, layering, locality, painting, perception, stacking, texturing

Abstract

This document describes four projects that, taken together, serve as a proof by example of the efficacy of a perception-motivated strategy for making graphics tools. More specifically, this strategy involves first selecting local features motivated by human perception and domain understanding, and then building algorithms that allow users to interactively edit these features. While the tools presented herein range over the domains of painting, compositing, and animation, all share a common philosophy – they spring from the isolation of a basic task-relevant feature and the creation of support algorithms that allow local edits of this feature at interactive rates.

Gradient Paint is a paint program inspired by theories of color perception which allows artists to edit edges between colors rather than colors themselves. The system includes a new real-time integrator for image display and a set of brushes suited for working with edges.

Local Layering introduces a notion of local overlap which allows artists to stack images as if they were paper cut-outs. In this work, I present a new way of representing stacked images and provide a provably correct and sufficient set of operators for navigating this representation.

Soft Stacking is a continuous-domain extension of Local Layering which allows artists to stack images as if they were volumes of fog. I present both brush-based and optimization-based creation techniques for such continuous-domain stackings.

3D-like Texturing for 2D Animation is an attempt to provide 2D artists with a way to reap one of the primary benefits – coherent texturing – of creating 3D models. The system I describe, though imperfect, is able to construct a correspondence between various 2D drawings using only local shape information, and in such a way that the computer never creates a 3D model.

These four projects showcase a new, and somewhat effective, strategy for creating graphics tools that transcend traditional art materials and computer constructs.

Contents

Chapter 1

Introduction

In the past 30 years, computer-based creative tools have gained immense sophistication. From simple, costly, hand-made framebuffers, we've moved to commercially produced graphics accelerators with on-board computing ability. From crude light pens, we've moved to reliable, pressure- and tilt-sensitive tablets. And from rack-filling PDP-11's, we've moved to orders-of-magnitude faster (and smaller) personal computers. Users routinely capture and manipulate images with tens of millions of pixels, and design 3D models with millions of polygons.

But while the sophistication of our hardware and algorithms has surged forward, the metaphors that guide our data representation have changed quite slowly. In many cases, artists seem to be restricted by tools that were designed more for machine than human – tools that fail to match the user's perception and understanding of a medium.[1]

As a small step toward updating these representations, at least in the discipline of image editing, I present in this document a proof-by-example of my thesis:

> New image editing tools may be created, and existing ones improved, by allowing interactive edits of perception-motivated local features.

Though this positive form of my thesis statement focuses on what we should include in image editing tools, there is an equally interesting negative corollary: image editing tools should preserve ambiguity – both task-irrelevant ambiguity (don't force the artist to specify unnecessary information) and intent ambiguity (don't infer more from an artist's action than necessary).

Taken together, my thesis and this corollary begin to sound almost like common sense. When creating a composition, artists should have tools that allow them to specify features relevant to the experience of the composition without worrying about defining irrelevant aspects. Tools should be predictable and minimal, free from magical side-effects and unexpected excitement. Though this sounds straightforward, few graphics

[1]E.g. raster-based paint programs limit the detail of an image to an arbitrary resolution; shoving faces together in a polygon-based modeler does not cause them to merge like clay; and layers in a compositing program cannot weave, even if the objects they picture could.

tools today follow such a philosophy. I expect this arises from two basic difficulties:

1. Relevance is generally task-dependent and hard to define, so "sufficient" is often used as a proxy.
2. Computers hate ambiguity.

In my prototype systems, which I will describe shortly, I address the question of relevance by appealing to perception (both established literature and self-experimentation) and the problem of ambiguity with algorithms designed to – when necessary – interpolate and extrapolate smoothly from user edits.

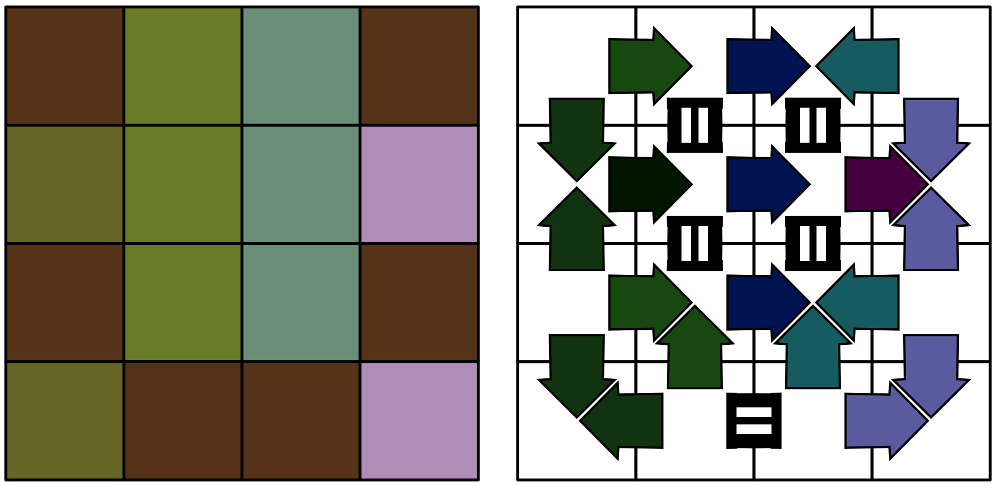
I will now briefly summarize the four new image editing systems which I have designed with this thesis and corollary in mind:

- In traditional paint programs, images are represented as grids of intensity values, like paint on canvas. However, perceptual studies indicate that the human visual system works on intensity differences. With *Gradient Paint* I allow artists to edit color differences directly, and leave absolute pixel color ambiguous (to be determined, later, by context).
- In standard digital compositing, graphical elements are ordered in global lists, like stacks of film. This prevents artists from achieving orderings that their intuition about physical materials tells them should be plausible. With *Local Layering* I allow artists to stack and weave graphical elements as if they were paper cut-outs; and with *Soft Stacking* I go further and let them stack and mix elements as if they were volumes of fog. Edits are local and do not constrain the global order of layers more than necessary.
- When drawing in a conventional 2D program, artists are restricted to tools that treat the drawing as a 2D object, even though the artist is often thinking of a shape with volume and 3D structure. With *3D Texture Mapping for 2D Drawing* I let artists specify the local structure of 3D surfaces using gizmos that describe local surface shape. These gizmos allow the artist to specify only as much surface detail as needed to transfer surface detail between frames, instead of building a full 3D model.

While the remainder of this document is concerned with changing the basic artist tools in the scenarios listed above, these are but a few instances of a wider opportunity in graphics – hardware and algorithms have advanced to the point where we can try new, alternative primitives to enable creativity, communication, and understanding.

In the remainder of this chapter, I will provide more detail about each of my selected problems, describing the origin of the tools I will be replacing, my motivation for doing so, and the consequences of the replacement. In each case the replacement will entail additional algorithmic effort, but will be rewarded by new creative capability. The overviews in this chapter are written to be informative but not overly sticky with the molasses of detail – for that, the reader may refer to subsequent chapters, which are liberally sweetened with the stuff.

1.1 Painting with Edges



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.

1.1.1 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.

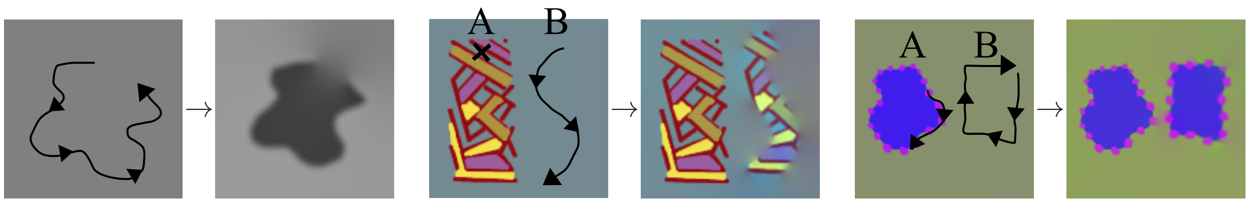


Figure 1.1: *The three paint brushes supported by Gradient Paint.* **Left:** *the gradient brush draws edges.* **Middle:** *the clone brush copies edges.* **Right:** *the edge brush copies and re-orients edges.*

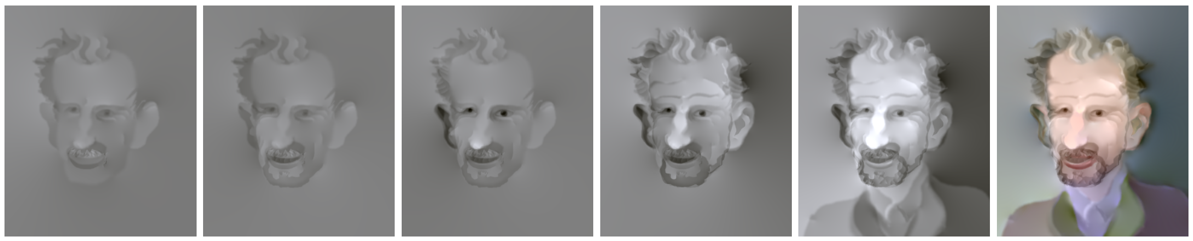


Figure 1.2: *Creating a new image by sketching with color differences. (Artist: M. Mahler.)*

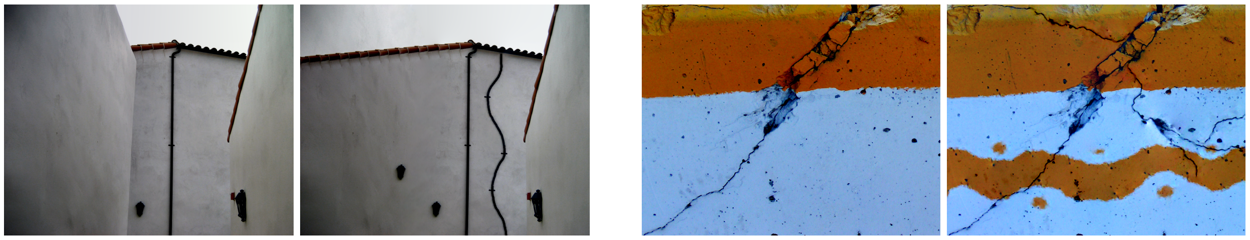
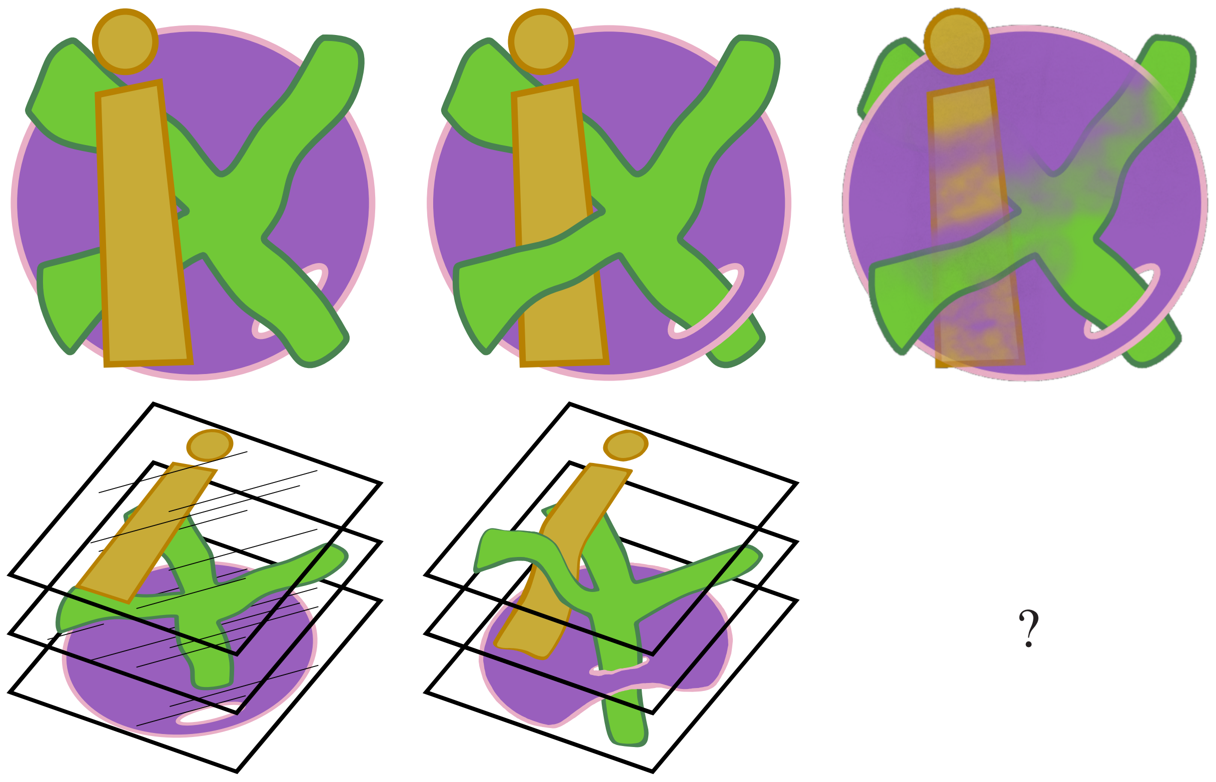


Figure 1.3: *Copying edges within an image to change a roofline and add more cracks. (Originals, leftmost in each pair, by clayjar (***left***) and Tal Bright (***right***) via flickr.)*

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.

1.2 Stacking Images Like Paper and 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 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).

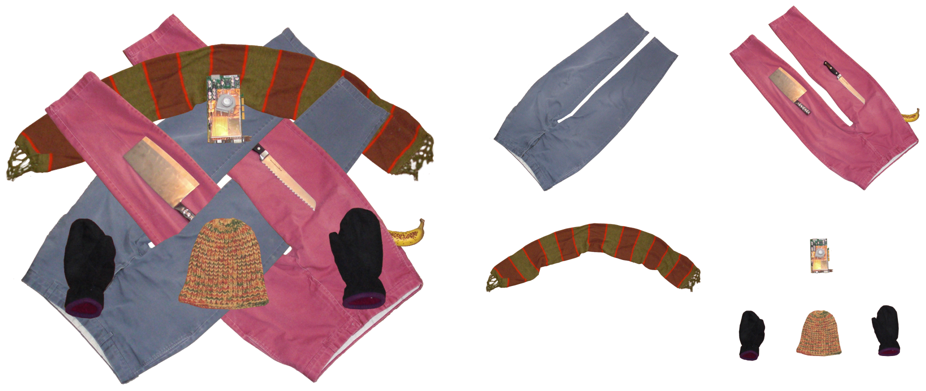
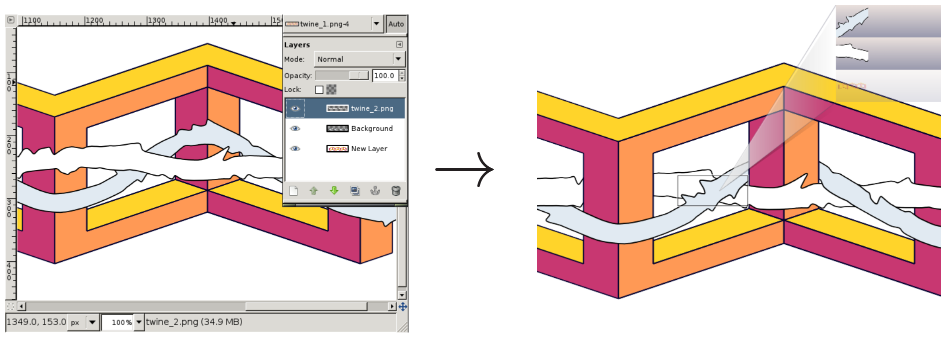


Figure 1.4: *A complex stack of four layers.*



Mode: Normal
Opacity: 100.0
Lock:
1349.0, 153.0 px
100%
twine_2.png (34.9 MB)

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

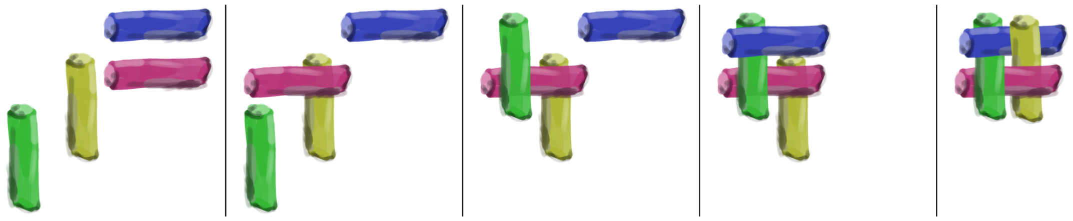


Figure 1.6: *Weaving layers by dragging them over each other.*

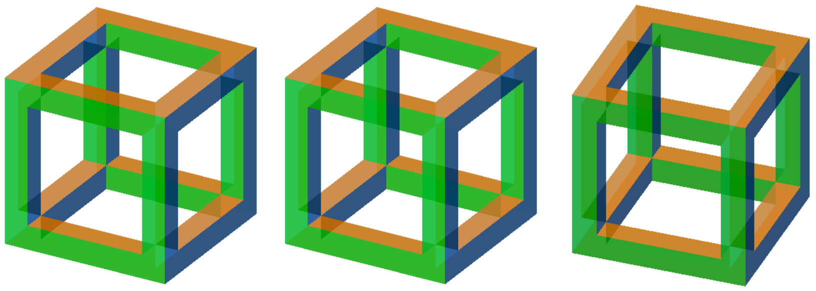


Figure 1.7: *An impossible figure created by locally changing depth order of a 3D model.*

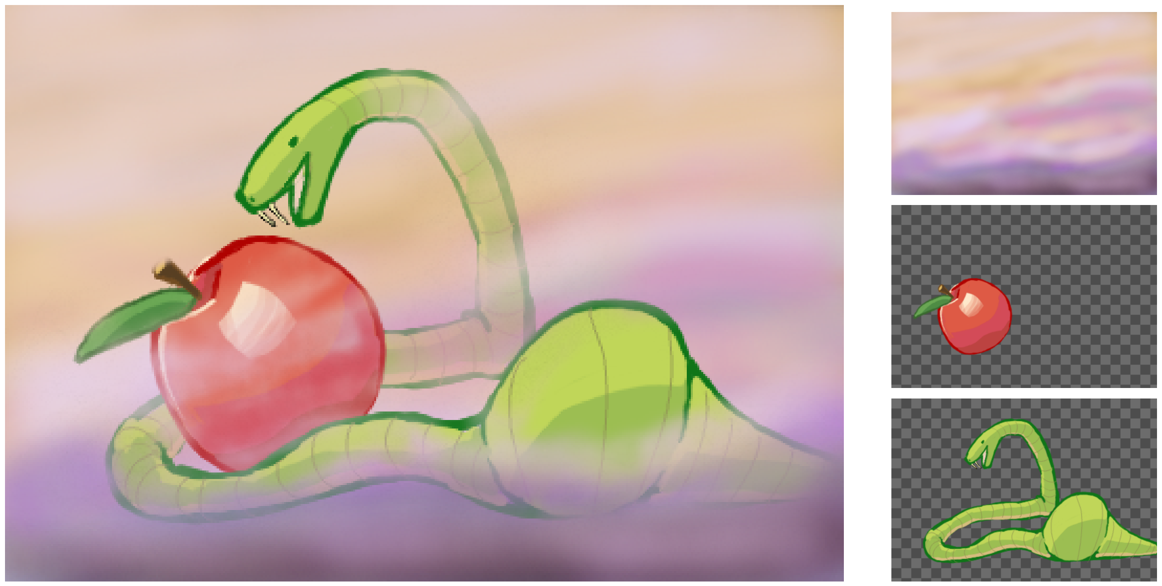


Figure 1.8: *Soft Stacking allows partially-transparent things (like mist) to gently mix with other layers. The artist specified the interleaving of fog, snake, and apple by painting local stacking orders.*

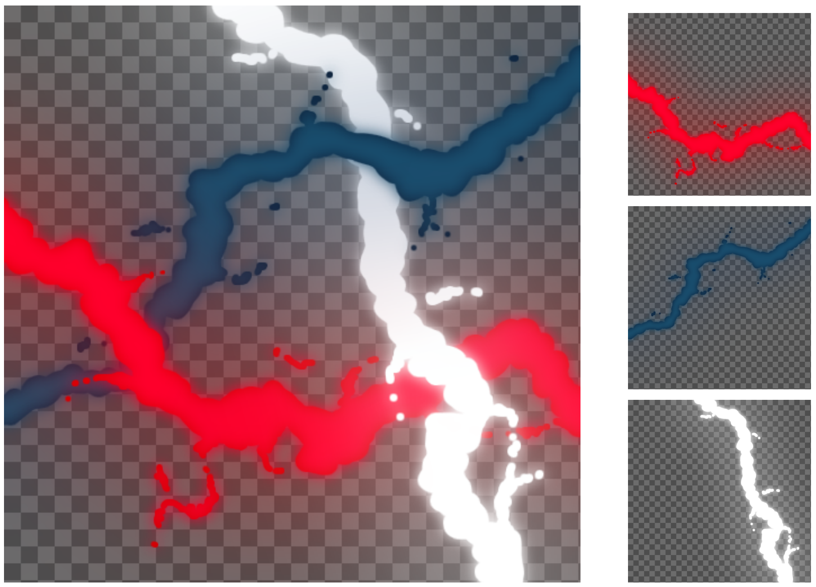


Figure 1.9: *In this Soft Stacking result, the artist specified constraints at the intersections of the bolts and the system solved for an optimal global order. Note the smooth mixing of the halos around the bolts.*

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.

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).

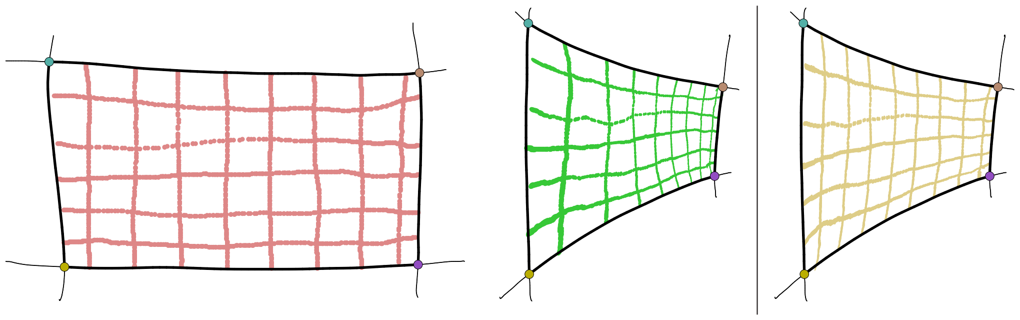


Figure 1.10: Left and middle, *Artist-specified shape cues allow for perspective-like effects in texture transfer. (Compare to,* **right**, *the same transfer with no shape cues.)*

[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.

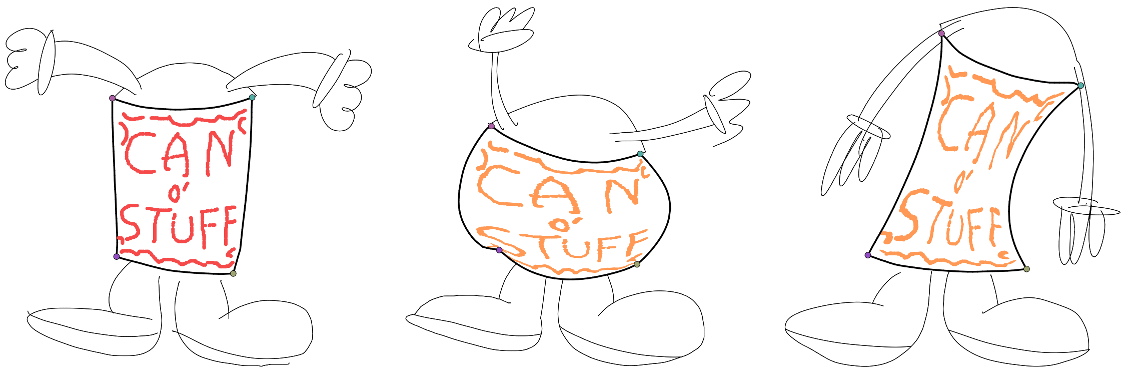


Figure 1.11: *My 3D-like texturing method works on deforming objects.*

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.

Chapter 2

A Brief and Biased History of Computer Graphics Tools

The motivation for my thesis – the reason such a proof-by-example is important – is that we stand at a point where plentiful hardware resources and decades of refinement and testing have allowed tool-makers to nearly perfect tools for most bread-and-butter graphics tasks. This section focuses on two important trends in this development process: the shift from building hardware to writing software, and the emulation of traditional art materials.

This is not a complete or unbiased history, and one should bear in mind that I am not a detached observer. However, it is my goal to give a sense of the order of events which led to the current state of affairs in tool design. It is also, somewhat, an attempt to characterize two design philosophies underlying many of today's graphics tools.

Additionally, this section is not intended to chronicle fine-scale related work for the tools presented in the remainder of the document. Such a unified accounting for such disparate research would be scattered at best. Thus, I defer accounting for the detailed research neighborhoods of each tool to their respective chapters.

2.1 From Hardware to Software

We begin with Sketchpad, the landmark computer-based drawing system designed by Ivan Sutherland and – in essence – the first computer art tool [Sutherland 1963]. Sketchpad was a computer-based drawing system. Today, that would mean it was a piece of microcode that ran on any number of common platforms. In 1963 that meant that it was a specialized computer (the TX-2) modified with a specialized display, specialized input devices, and custom software to tie things together (Figure 2.1).

The story of how we progressed from a hardware-software discipline to an almost entirely software discipline is part of the story of computer graphics. We would not have achieved the present level of sophistication in today's graphics tools if every researcher, like Sutherland, needed to re-invent and re-implement basic

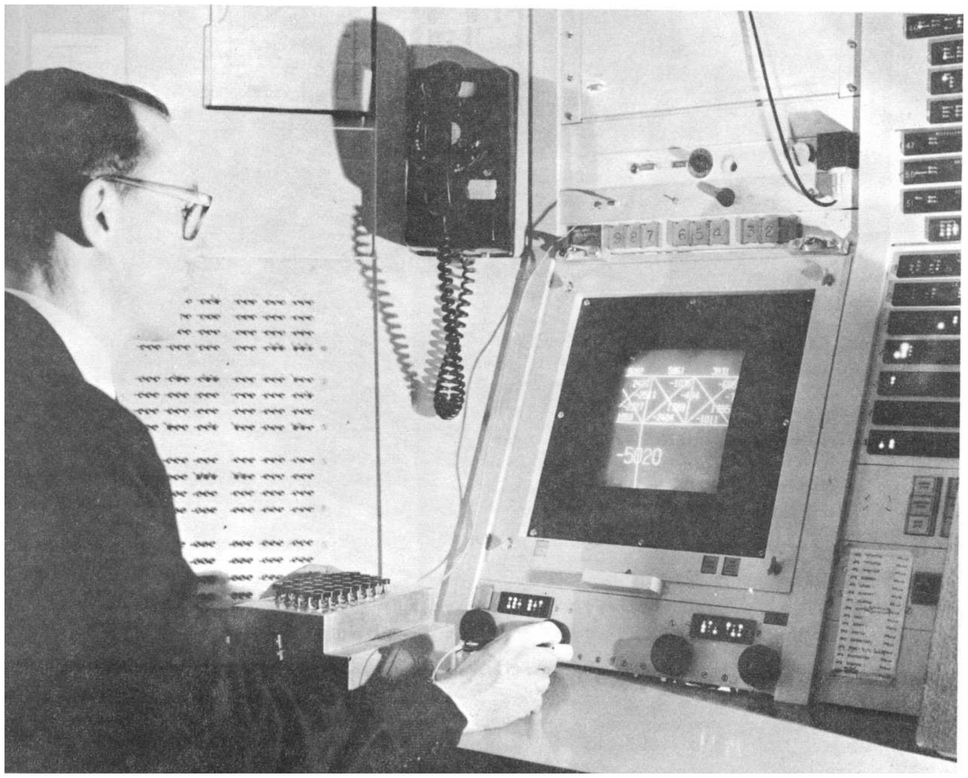


Figure 2.1: *Ivan Sutherland's Sketchpad. Picture from Sutherland [1963].*

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 NTSC television.

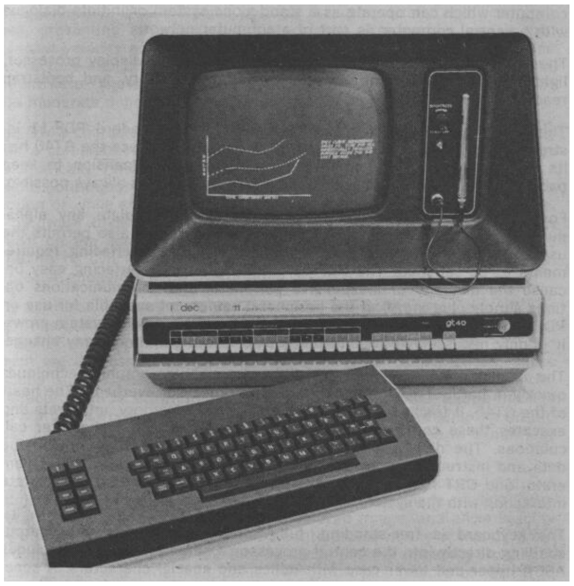


Figure 2.2: *A DEC GT-40 terminal, introduced in 1972, features many of the same I/O capabilities as Sutherland's Sketchpad system including vector-based graphics and a light pen input device. Image from the 1976 PDP-11 Peripherals Handbook, as appears at http://www.brouhaha.com/ eric/retrocomputing/dec/gt40/handbook.html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

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.

EGA, or VGA specifications could be expected to run on a variety of IBM PC (and cloned) hardware. While this meant that graphics programs could operate in a uniform environment, the interface to these graphics adaptors was no more sophisticated than the framebuffers of the 1970s; one set some hardware registers to select a display mode, then wrote data into a special region of memory to be interpreted as color information and displayed.

It was during the 1990s that this basic interface to graphics hardware would finally be challenged. Sophisticated GUI-and-window operating environments both substantially increased the number of graphical operations required and provided (implicitly via standard GUI toolkits, or explicitly via drawing libraries) the layer of abstraction required to support hardware acceleration of common 2D graphics tasks. Similar drawing libraries were devised for textured-triangle 3D graphics[2], though it was not until the latter half of the nineties[3] that one could expect these to be more than interfaces to high-performance software renderers on consumer-level computers.

As originally specified, these graphics APIs assume a fixed-function accelerator that generates, textures, and composites triangles subject to a few user-selectable options pertaining to lighting and blending; and, indeed, this is what early 3D accelerators did. The rigid operational specification allowed designers to allocate most of the accelerator's functionality to carefully designed hardware for maximum speed.

In the 2000s, however, faster chips allowed much of this dedicated hardware to again be implemented in software – software running on special processors on the graphics accelerator itself. This meant that accelerators could be programmed to adapt to individual workloads, and even perform general-purpose computation. The OpenGL and Direct3D APIs have been continuously adapted to this changing platform and now much more resemble a programming interface for a quirky sort of data-parallel computer than a specific set of graphical algorithms.

Such general-purpose drawing hardware leaves a lot of open space to innovate in. And so the research community continues to ponder drawing primitives (because, for many tasks, a bunch of triangles isn't the ideal representation) and hardware acceleration (because, well, more speed and more flexibility couldn't hurt). One finds a snapshot of these two threads in the proceedings of the most recent (as of this writing) SIGGRAPH conference: a programming system for motion effects [Schmid et al. 2010], a way of defining 3D shapes with silhouettes [Rivers et al. 2010a], and a more effective rasterization approach for graphics processors [Fatahalian et al. 2010] are all present.

And so it is that after forty years of computer graphics, we are still trying to figure out exactly what our basic hardware and software in support of computer graphics should be, beyond general enough to do whatever we happen to want at the moment and fast enough that it gets done quickly. All told, I don't think this is too bad a place to be.

[2]SGI's OpenGL, standardized in 1992, was based on the graphics framework used in its workstations; Microsoft's Direct3D, released in 1996, was part of an effort to make game programming in Windows 95 more appealing.

[3]The first high-performance consumer-level 3D accelerator was 3dfx's Voodoo PCI, released in 1996.

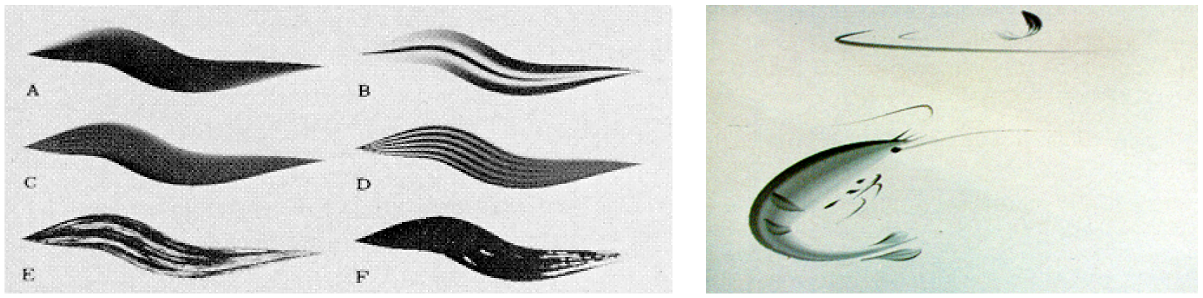


Figure 2.3: *Example strokes and a completed illustration with simulated brushes. Figure from Strassmann [1986].*

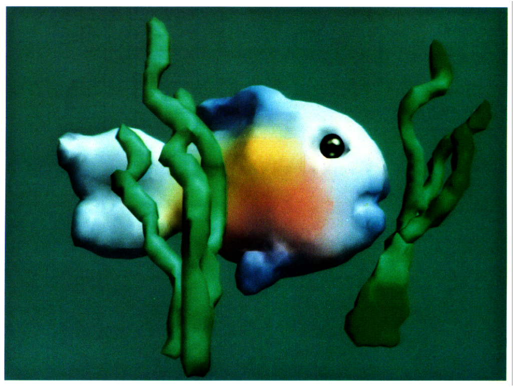


Figure 2.4: *An example sculpture presented in Galyean and Hughes' 1991 paper.*

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 honor probably belongs to a muddied hand or chalky rock.

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

Similarly, in 3D modeling, early computer-based tools relied on mathematical definitions of objects that were convenient for display through ray-tracing or scan conversion. Inspired by traditional techniques, graphics researchers brought sculpting tools to the computer [Galyean and Hughes 1991] (Figure 2.4), even going so far is to simulate clay and provide haptic feedback [McDonnell et al. 2001]. And, eventually, commercial software emerged that provided a sculpting-like experience for 3D models [Pixologic 2004].

But why have computer graphics researchers been using computers to emulate these existing art tools, rather than finding dramatic new ways of making art with them? I think the answer, in part, is one of mindset. The typical engineer-minded computer programmer is trained to think about refining real-world processes in a structured way: the programmer will examine the process, extract a specification, and implement the specification efficiently. So, for instance, a systems programmer may look at a travel agency's airline booking process and note that the by-hand lookup of flight times could be replaced by an automated lookup. She would then construct a specification for flight lookup based on her knowledge of the travel agency, perhaps check this specification with one of the agents, and finally produce a computer system that implemented the required functions.

This computer engineering mindset, when applied to art, results in natural media paint tools.

Such tools are not inherently bad. Simulation of real-world art materials can add detail and life to what otherwise might be an entirely mechanical process; and bringing these tools into a computer allows for more exploration and experimentation (because of the ability of artists to save various versions of a work). However, computers can do more than copy real-world art materials.[6] Strassmann, the author of the original Hairy Brushes system and, to some extent, progenitor of the natural media painting movement, puts it well in his 1986 paper:

> Occasionally, when I am asked about the limits of realism in my simulation, I am reminded of similar questions asked of builders of electronic instruments. The answer, of course, is that there is room in electronic media for both accurate reproduction of physical phenomena, and for creative exploration with totally new forms of expression which take advantage of the differences inherent in the new media.

2.3 So Now What?

And that, roughly, brings us to the present day. The reason I've chronicled these two story arcs in computer graphics is to shed light on the design of today's mainstream computer-based art tools.

The relative scarcity of computational resources during the birth of graphics shaped these tools just as much as any philosophy or user study could. The basic algorithms and data structures used were those that fit in limited memory, or were compatible with a limited display system. Many of these data structures persist to this day; worse still, these structures are often exposed directly to the user – paint programs allow one to edit individual pixels, and many 3D modeling programs involve directly manipulating triangles. This further hampers development because the antiquated inner arcana of our tools has become a user expectation.

[6]And the market for art tools seems to bear this out – both commercial programs I mention face strong competition from other tools with very few natural media features.

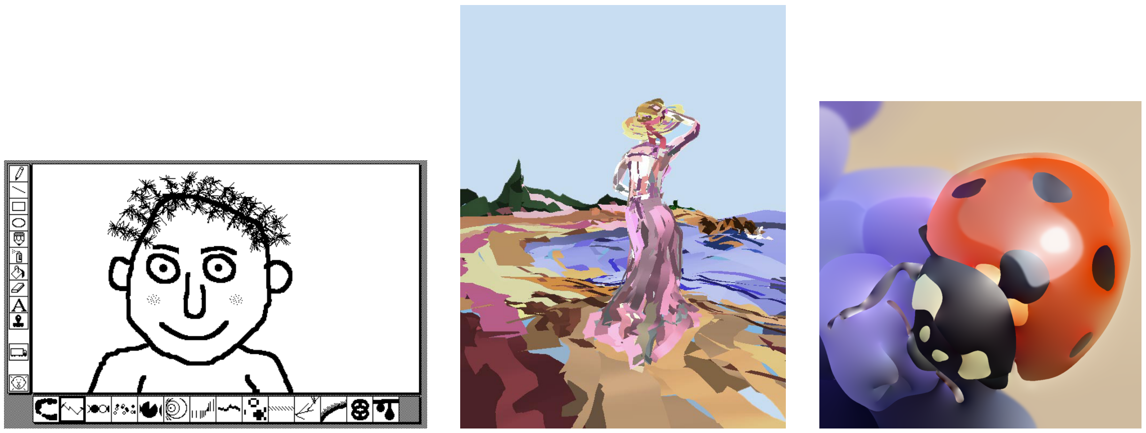


Figure 2.5: *Three graphics tools that are neither copies of real-world art materials nor consequences of limited hardware.* **Left**, *Kid Pix is a wacky paint program.* **Center**, *CavePainting is a combination of painting and sculpting, using 3D brush strokes.* **Right**, *Diffusion Curves smoothly interpolates color constraints. (Images from Hickman [1989], Keefe et al. [2001], and Orzan et al. [2008].)*

Then there are those computer graphics tools that insist on further following the artist's intuition by copying real-world art supplies. This is certainly a win in the short term – the artist can bring his real-world skills to a computer program and achieve the same effects, with the added ability to undo it at will. This power provides a lot of freedom, but not nearly so much freedom as we've taken away by limiting the artist to centuries-old implements built for the expediencies of a physical world.

So this is where tool design sits today. There are legacy algorithmic constructs – hacks to make software that works with limited resources – which are no longer needed, but their prevalence (and user-visibility) gives them staying power. There are those who imitate physical art media – often to good effect – but this constrains them to well-explored creative territory.

This is a shame, because there is more computing power and more background data and more inspiration to be had today than at any point in the past. And there *are* graphics tools out there that defy these trends (Figure 2.5): for example, CavePainting is a combination of painting and sculpture which could not exist outside of a computer [Keefe et al. 2001]; Diffusion Curves put general smooth color interpolation into the hands of artists [Orzan et al. 2008]; and Kid Pix is a paint program for children with dozens of wacky tools [Hickman 1989]. In subsequent chapters I will describe four more projects in this exclusive cadre. These projects are distinguished[7] by their motivation: intuitive editing of relevant local features and ambiguity preservation. However, my methods remain radical departures from both algorithmic convenience and historical convention.

[7]Distinguished from CavePainting and Kid Pix, that is. Diffusion Curves shares much of the same motivation as my Gradient Paint project.

Chapter 3

Gradient Paint

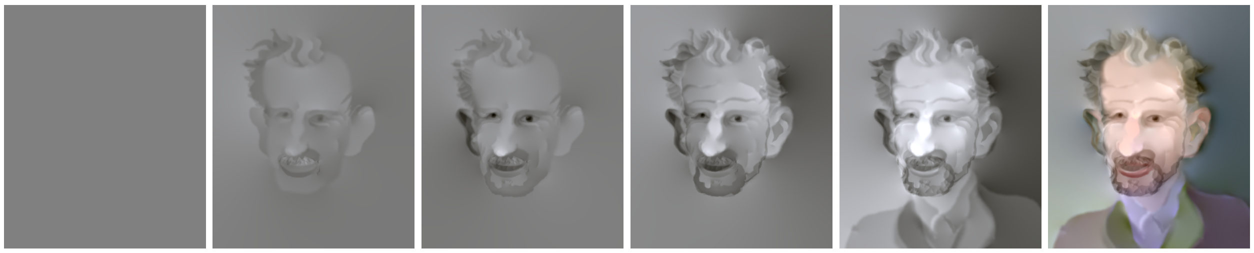


Figure 3.1: *Starting from a blank canvas, an artist draws and colors a portrait using gradient-domain brush strokes.* Additive *blend mode used for sketching,* directional *and* additive *for coloring. Artist time: 30 minutes.*

This section describes a system that allows artists to paint with color differences instead of color values (Figure 3.1). As required by my thesis, this system introduces an intuitive local feature (color differences $\approx$ image edges), and allows artists to interactively edit and create images with this feature. The system propagates these local edge edits smoothly, not introducing any additional edges, and thus preserves the ambiguity of the artist's intent.

Working with color differences (edges) in images is an old idea in graphics and vision. This representation is motivated by the perceptual work of Land and McCann[1] [1971] which, in turn, draws upon much earlier ideas about the human visual system [Werner 1935]. As a sampling, gradient-domain (i.e., edge-based) methods have been used to represent and edit images, composite images, remove shadows, and perform tone-mapping.

However, while these techniques do allow sophisticated image manipulations at interactive rates, they do not allow the creation of wholly new images. That is, there is no existing facility for artists to paint directly in the gradient domain. To do this, my system needed two things: a fast[2] method of going from pixel-color-

[1]no relation

[2]Real-time, i.e., 20fps, instead of interactive, i.e., 5fps.

differences (an overcomplete representation) to pixel colors, and a set of paint brushes for the artist to use. For the former, I used a GPU-based multigrid, similar to Goodnight et al. [2003] and Bolz et al.'s [2003] work. For better convergence, I performed an eigenvalue analysis of the multigrid v-cycle operator to numerically optimize certain solution coefficients. The brushes (and the notion of gradient domain painting), however, are the main contribution of the work.

3.1 Contribution

The contribution of this work was to demonstrate gradient-domain image manipulation with real-time brushes. (Previous approaches generally operated on selections or pasted regions at interactive, but not quite real-time, rates.) I introduced three brushes:

The Gradient Brush, with which one may paint paint color differences directly. This was a simple brush to implement, but it is also represents the striking revolution at the core of Real-Time Gradient-Domain Paint: with this brush, you can create new images. All previous gradient-domain editing projects had been about editing images.

The Clone Brush, with which one may copy portions of the image around. This brush was essential, as seamless copy-and-paste has been shown to be a strength of gradient-domain approaches by previous work.

The Edge Brush, with which one may copy and re-orient edges in the image. This brush is my take on what a gradient-domain clone brush should be. Since so much of gradient-domain editing is taken up with edges, it seems like the right way to clone is to copy edges of regions. This brush allows one to draw a line in the image, the edges near which are replayed when drawing with the brush.

3.2 History

In 1971, Land and McCann[3] published an article describing a theory of color perception motivated by a series of remarkable experiments. In these experiments, an arrangement of colored paper squares were illuminated by controllable short-, mid-, and long-wave lights (that is, blue, green, and red lights). These squares were illuminated at some default lighting condition and the light (of each color) reflected by a white square was measured. Various other lighting conditions were then selected such that the light reflected by other, colored, squares was the same as that reflected by the white square in the original condition.

Despite the fact that in these variant lighting conditions the distribution of light reaching the eye from a yellow, blue, lime green, or grey square was the same as that reaching the eye in the original condition from a white square, subjects reported that the overall color sensation remained constant. Therefore, Land and McCann concluded, color perception must involve estimation of surface reflectance (not just measurement of incident light).

[3] still no relation

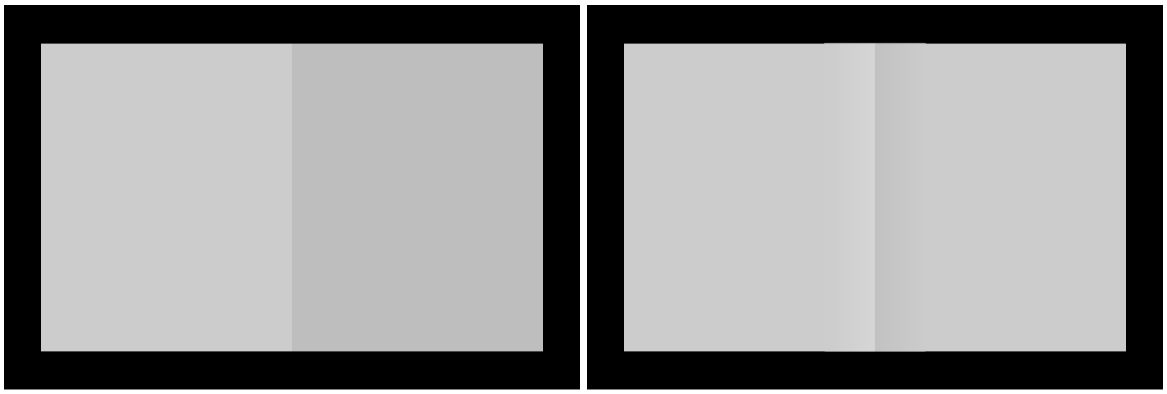


Figure 3.2: Left, *"two squares and a happening", adapted from Land and McCann's illusion. Place a pencil or other thin object across the boundary between the squares and their apparent lightness becomes equal.* **Right**, *a similar illusion devised by Cornsweet. The right square appears darker due to strong edge in the center of the image (even though most of the area of the square is the same lightness as the left square).*

They further propose an algorithm ("the retinex algorithm") for performing this surface reflectance estimation by considering lightness ratios at edges between squares. This use of edges is motivated – in part – by their "two squares and a happening" illusion, a version of which is presented in Figure 3.2, along with another similar illusion. In this illusion, two squares of different apparent lightnesses become the same shade when a pencil or other object obscures the edge between them.

While the specifics of the retinex algorithm may not exactly match the stated goals [Brainard and Wandell 1986], the motivating experiments provide compelling evidence that color differences are a far more important visual feature than absolute colors. This evidence has not escaped the purview of graphics researchers.

The first instance of edge-focused image editing was the contour-based image editor of Elder and Goldberg [2001]. Their high-level edge representation enables users to select, adjust, and delete image edges. These interactions, however, are far from real-time; users queue up edits and then wait minutes for the results. The paper describing this section of my thesis appeared concurrently with the work of Orzan et al. [2008]; they describe a vector graphics color-filling technique based on Poisson interpolation of color constraints and blur values specified along user-controlled splines – a representation similar in spirit to Elder and Goldberg's edge description. A GPU-based multigrid solver (similar to my own) and a fast GPU-based blur computation enable interactive editing of this representation.

Classic gradient-domain methods include dynamic range compression [Fattal et al. 2002], shadow removal [Finlayson et al. 2002], and image compositing [Levin et al. 2003; Pérez et al. 2003]. (One key to seamless image stitching is smoothness preservation [Burt and Adelson 1983], which gradient methods address directly.) A survey of gradient-domain techniques can be found in the ICCV course of Agrawal and Raskar [2007].

Broadly, current gradient-domain techniques would appear in the "filters" menu of an image editing program, performing a global operation automatically (range compression [Fattal et al. 2002], shadow removal [Finlayson et al. 2002]), or in a way that can be guided by a few strokes (color editing [Pérez et al.

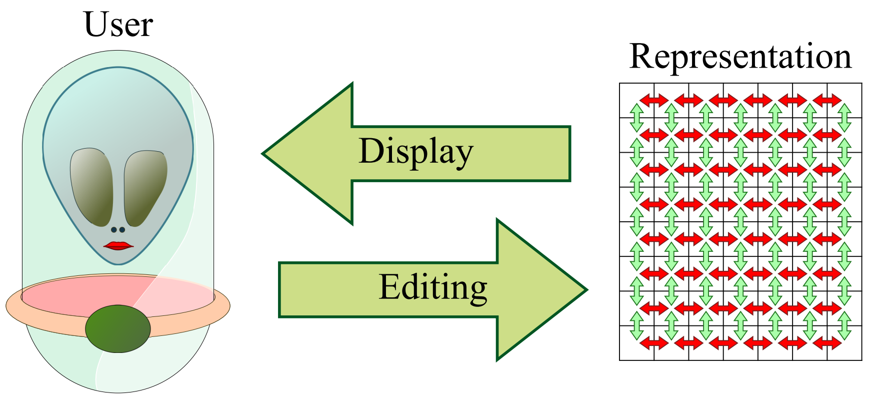


Figure 3.3: *A high-level system diagram for Gradient Paint. The user may manipulate a pixel-difference image representation (Section 3.3.1) using a variety of brushes (Section 3.3.2), viewing the resulting image through a real-time integrator (Section 3.3.4).*

2003]) or clicks (compositing [Levin et al. 2003; Pérez et al. 2003]). For the seconds-long integration processes used in these techniques, it is important to have such sparse input, as the feedback loop is relatively slow.

Recently, interactive performance was demonstrated for tone-mapping operations on 0.25-megapixel images [Lischinski et al. 2006]; this performance was obtained through preconditioning, which – due to the high calculation cost – is only suitable for edits that keep most of the image the same so that the preconditioner may be reused. A different preconditioning approach is presented in Szeliski [2006].

Agarwala [2007] addressed the scaling behavior of gradient compositing by using quadtrees to substantially reduce both memory and computational requirements. While his method is not real-time, it does scale to integrating substantially larger gradient fields than were feasible in the past – with the restriction that the answer be smooth everywhere except at a few seams.

Perhaps surprisingly, no great breakthrough in technology was required to provide the real-time integration rates at the core of my program. Rather, I simply applied the multigrid method [Press et al. 1992], a general framework for solving systems of equations using coarse-to-fine approximation. I use a relatively straightforward version, overlooking a faster-converging variant [Roberts 2001] which is less suitable to hardware implementation. My solver is similar in spirit to the generalized GPU multigrid methods presented by Bolz et al. [2003], and Goodnight et al. [2003] though some domain-specific customizations (e.g., not handling complicated boundary shapes) make it both simpler and somewhat faster.

3.3 System

This section presents a system which enables artists to paint in the gradient domain much like they are accustomed to painting in the intensity domain. I will describe this system in three parts (Figure 3.3): first, I describe the internal image representation used by the system; then I describe the brushes and blending modes my system provides that allow the artist to manipulate this representation; finally, I describe the real-time integrator that allows the system to display the result of these manipulations to the artist in real time.

3.3.1 Representation

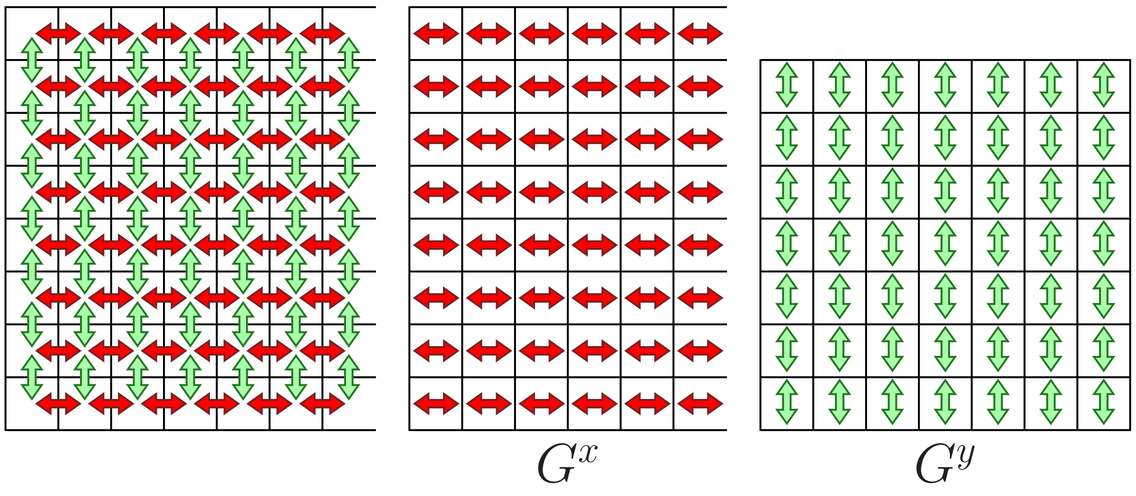


Figure 3.4: *Gradient Paint represents an image u by the desired color differences between pixels (**left**). These differences are stored in two arrays, G^x for x differences (**middle**) and G^y for y differences (**right**). These are only desired differences because, after editing, there may be no image u which exactly respects both G^x and G^y.*

Gradient Paint, taking a cue from other recent edge-based graphics tools (e.g., [Pérez et al. 2003]), stores images as dense desired x and y gradients (Figure 3.4). That is, for every adjacent pair of pixels it stores a desired difference between that pair. Thus, my system represents an $X \times Y$ image as an $(X-1) \times Y$ array of desired differences in the x direction and an $X \times (Y-1)$ array of desired differences in the y direction. I call the array of desired x differences G^x and the array of desired y differences G^y.

This representation allows the system to operate directly on color differences. However, it also introduces some subtleties in editing (since the x and y differences are sampled at different locations) and complexity in display (since the desired x differences and y differences may not always agree). These issues will be discussed in the following sections.

3.3.2 Brushes

In Gradient Paint, the user manipulates images with a set of brushes. Like brushes in an regular paint program, these brushes perform local edits of the underlying image representation. Part of the power of working in the gradient domain is that these local (edge) edits can actually produce non-local changes in image intensity.

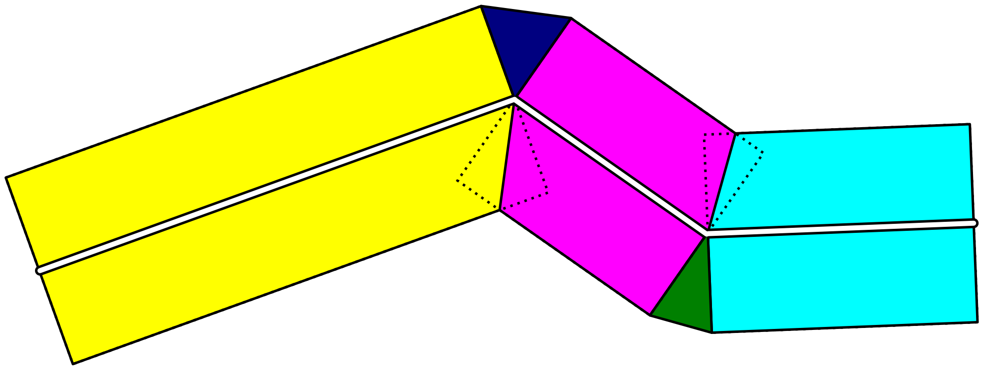


Figure 3.5: *In my programs, strokes are composed of chains of quadrilaterals around a polyline spine, with gaps filled by triangle-shaped wedges and overlaps resolved by selecting the quadrilateral with the nearest spine segment.*

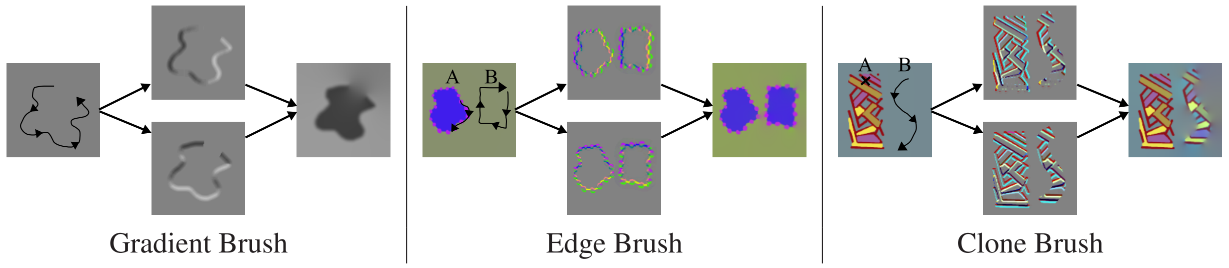


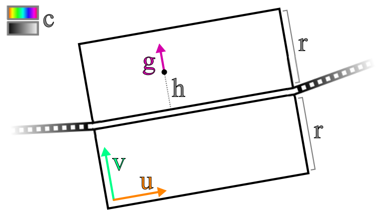
Figure 3.6: *Operation of the system's brushes illustrated by original image and stroke(s), produced gradients, and final image. My system provides real-time feedback during stroke drawing. The* **gradient brush** *paints intensity differences. The* **edge brush** *allows selection (A) and playback (B) of an edge; the edge is looped and re-oriented to allow long strokes. The* **clone brush** *copies gradients relative to a source location (A) onto a destination stroke (B).*

An additional quirk of working in the gradient domain is that brush strokes must actually paint into two images (the desired x gradient, G^x, and the desired y gradient, G^y), both of which are sampled at slightly different points. This requires some interpolation when blending the stroke with the existing image; I will deal with this in the section on blending modes, below.

In my system, a stroke is defined by a polyline and a radius. The stroke is rasterized as a series of quadrilaterals (one per segment) with triangular caps at corners (Figure 3.5). When multiple quadrilaterals cover a pixel, the gradient from the segment whose spine is closest to the pixel is used. (As the prototype is written using OpenGL, this step is performed by rendering stroke segments as tent shapes and using the z-buffer to select the closest.)

At present, I will describe the operation of each of the three different brushes. For each brush I will first describe the effect of the brush from the user's point of view – illustrated in Figure 3.6 – and then provide a technical description of how the system computes a desired gradient and blending weight for a given point inside a stroke quadrilateral or triangular cap.

Gradient Brush



$$g \equiv cu \tag{3.1}$$

$$\alpha \equiv \frac{r - \frac{|h|}{r}}{r^2} \tag{3.2}$$

Figure 3.7: *The desired gradient at a given position a gradient brush stroke is the perpendicular vector to the spine multiplied by the current color. The blending weight at a given position increases linearly to the stroke's spine and is normalized by stroke radius.*

The **gradient brush** is my system's simplest gradient-domain paint tool. As a user paints a stroke with the gradient brush, the brush emits gradients of the current color perpendicular to the stroke direction and the stroke weight is set to increase linearly from zero at the outside edge of the stroke to $\frac{1}{r}$ at the spine (this normalizes for stroke radius). Equations are given in Figure 3.7. Sketching with the gradient brush, an artist is able to define volume and shading as she defines edges, without ever touching the interior of shapes. An example portrait created with gradient-domain sketching is shown in Figure 3.1.

Edge Brush

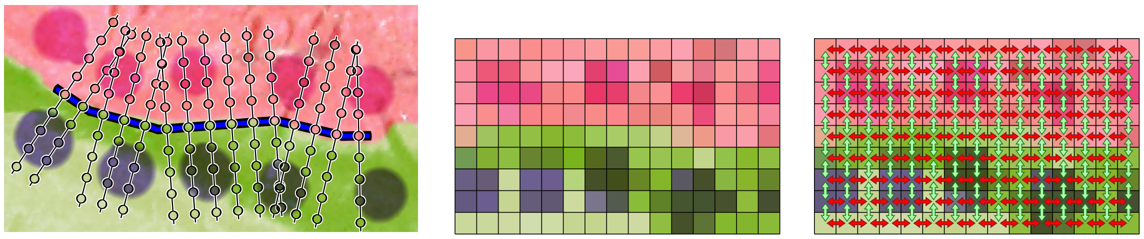
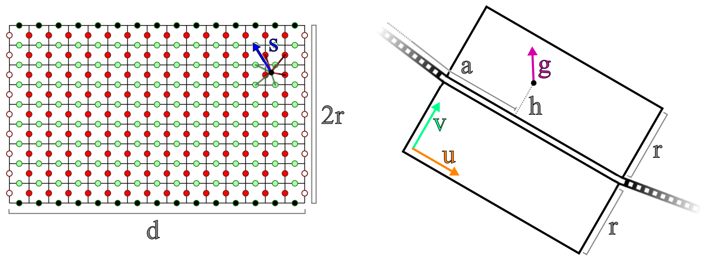


Figure 3.8: *Capturing the appearance of an edge. First,* **(left)**, *the displayed intensity image is sampled in 1-pixel wide strips perpendicular to the spine of the user's stroke. This produces an intensity image of the stroke's neighborhood,* **(middle)**, *from which gradients,* **(right)**, *are computed.*

The trouble with edges produced by the gradient brush is that they don't have the subtle texture and impact of natural edges found in real images. The **edge brush** is my simple and direct solution to this problem. The user first paints an edge selection stroke along a segment of an edge she fancies. The system captures the colors around this stroke and computes their gradients (Figure 3.8). Now, as the user paints with the edge brush, these captured gradients are "played back" – transformed to the local coordinate system of the current stroke and emitted (Figure 3.9). The system loops edge playback so the user can paint long strokes. Images whose production depended on edge capture and playback are shown in Figure 3.10 and Figure 3.11.



(3.3)
$$s \equiv \text{sample}\left(a - d\left\lfloor \frac{a}{d} \right\rfloor, h + r\right)$$

(3.4)
$$g \equiv s^x u + s^y v$$

(3.5)
$$\alpha \equiv 1 \text{ almost everywhere; see caption}$$

Figure 3.9: *Drawing with the edge brush. Emitted gradients are sampled from the previously captured edge-relative gradients and transformed into stroke-relative coordinates. The x-coordinate of the sample is determined by dividing the total distance along the stroke, a, by the length, d, of the sampled texture. This recycles the captured edge texture to allow the user to paint strokes longer than her originally captured edge. When sampling, gradient values are interpolated bilinearly, with extreme y-gradients (black points) assumed to be zero and extreme x-gradients (white points) taking the value of the nearest other x-gradient. The blending weight is one everywhere except in the pixels at the very edge of the stroke, where if falls off linearly to zero.*

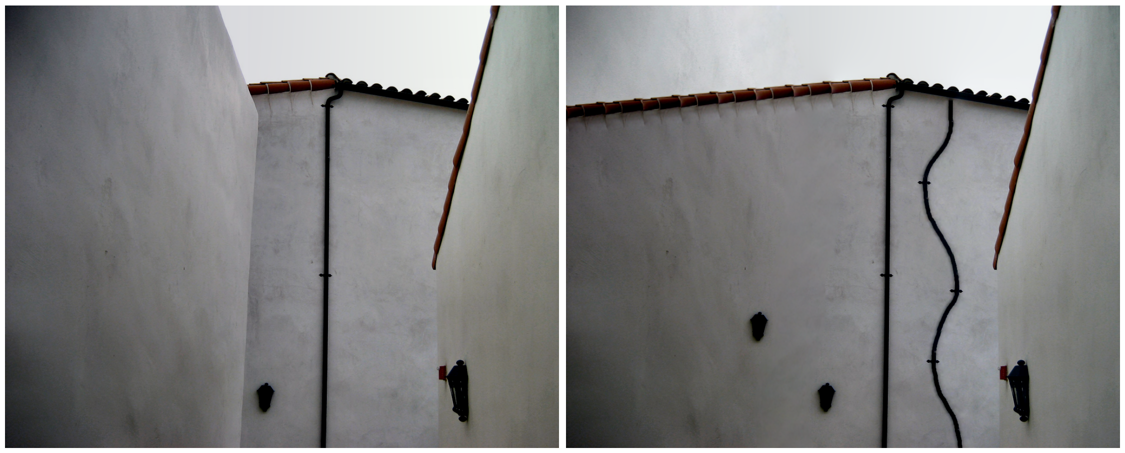


Figure 3.10: *Before* **(left)** *and after* **(right)** *editing with the edge brush. The user captured the smoothness of the left wall and used it to paint out the edge separating the left and back walls. The user then selected a segment of downspout and created a more whimsical variant. Finally, the tile roof-line to the left was captured and extended. Finishing touches like the second lantern and downspout cleats were added with the clone brush. The edge was painted out with* over *blend mode, while elements were added using* over *and* maximum *modes. Artist time: 2.5 minutes. (Original by clayjar via flickr.)*

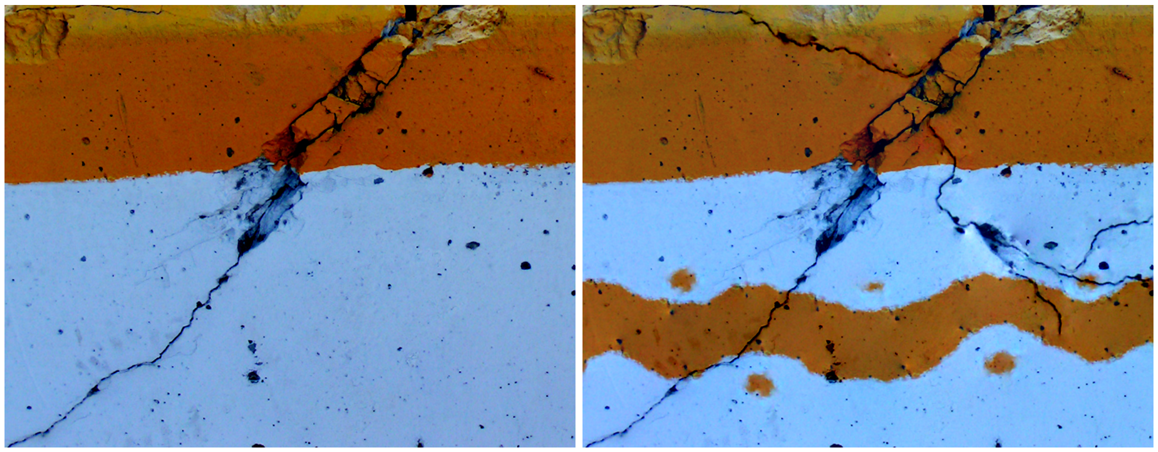
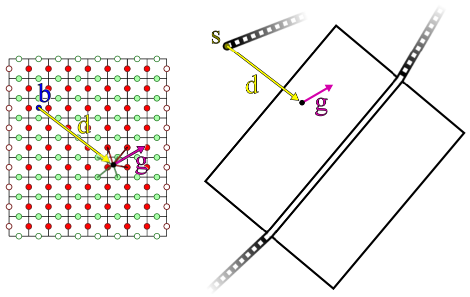


Figure 3.11: *Before* **(left)** *and after* **(right)** *editing with the edge brush. The user extended the crack network using a segment of existing crack and added her own orange stripe to the image with just three strokes – one to select the original edge as an example, and two to draw the top and bottom of the new stripe. Dots where then added by drawing small circles with the same captured color difference. All strokes use* additive *blend mode. Artist time: 3 minutes. (Original by Tal Bright via flickr.)*

Clone Brush



$$g \equiv \text{sample}\,(b + d) \tag{3.6}$$

$$\alpha \equiv 1 \tag{3.7}$$

Figure 3.12: *Drawing with the clone brush. Before drawing, the user sets a base point, b, in the image. The emitted gradient at a point with offset d from the start, s, of the stroke is sampled from the current desired gradients at image point $b + d$. Notice that, unlike the edge brush, copied gradients are not re-oriented to follow the stroke.*

Because image compositing is a classical and effective gradient-domain application [Pérez et al. 2003; Levin et al. 2003], I include it in my program. The **clone brush** copies desired gradients from one part of the image to another (Figure 3.12). By replacing copy-and-paste or cumbersome draw-wait cycles with a direct clone brush, I give the user more freedom, granularity, and feedback in the cloning process. In fact, because my integrator is global, the user may drag entire cloned regions to, say, better align a panorama – all in real-time. Cloning can be used to move objects, as in Figure 3.13 and Figure 3.14. Cloning with multiple blend modes allows copying an object and its lighting effects as in Figure 3.15. A more subtle clone effect is the skin texture adjustment in Figure 3.17.

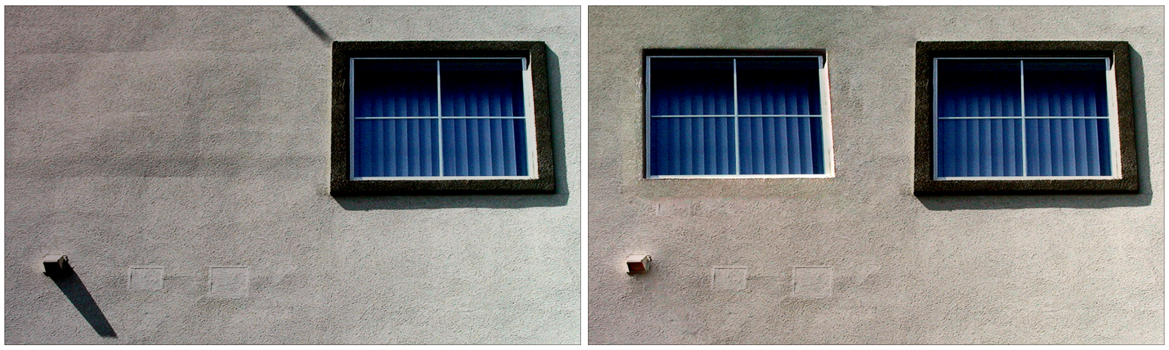


Figure 3.13: *Before* **(left)** *and after* **(right)**. *The left window is a clone of right (with* over *blend mode), with the outer edge of the surround omitted. Color adjustment (gradient brush with* directional *blend mode) was used to make the inside of the left window the same shade as the right. Two shadows were removed by cloning out edges. Artist time: 1.5 minutes. (Original by P Doodle via flickr.)*

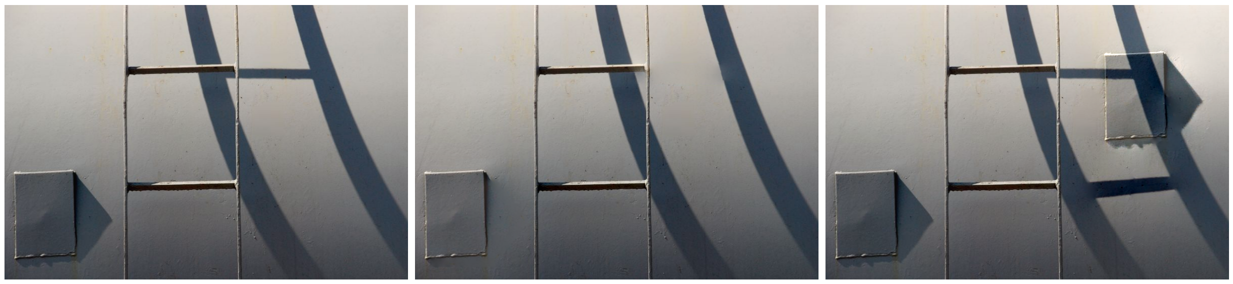


Figure 3.14: *Before* **(left)** *and two altered versions* **(middle, right)**. *Shadow adjustments and removal, cloning, and edge capture all come into play.* Over *blending used for erasing;* maximum, over, *and* additive *blending used for re-painting. (Original by jurek d via flickr.)*

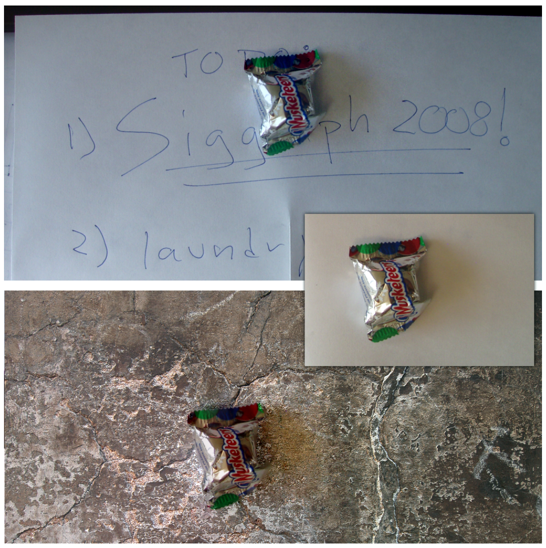


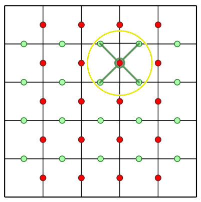
Figure 3.15: *Cloning with multiple blend modes. The candy* **(middle right)** *is cloned onto two different backgrounds* **(top, bottom)** *with* over *blending, while its shadow and reflected light are cloned with* additive *blending. Artist time: 4 minutes. (Bottom background by Stewart via flickr.)*

3.3.3 Blend Modes

Depending on the situation, the artist may change how her new strokes are blended with the background. Below, I describe the blending modes my paint program supports. Blending is a local operation; it uses some function B to combine the current desired gradient, $g \equiv (g^x, g^y)$, with the desired gradient painted by the user, $p \equiv (p^x, p^y)$, to attain a new desired gradient. The stroke weight, α, is used to interpolate between this new desired gradient and the current one.

This process is somewhat complicated by the fact that the desired x and y gradient arrays, G^x and G^y, are sampled at different locations in the image. This is resolved by running blending at x gradient locations (using interpolated y gradient values), and storing only the x component of the result, and running blending at y gradient locations (using interpolated x gradient values) and storing only the y component of the result. This is illustrated in Figure 3.16.

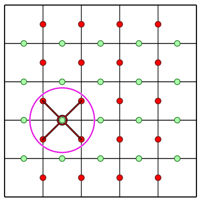
The same blending process is applied across all color channels.



$$g_{x+0.5,y} \equiv \left(G^x_{x,y} \; , \; \frac{G^y_{x,y-1} + G^y_{x+1,y-1} + G^y_{x,y} + G^y_{x+1,y}}{4} \right) \tag{3.8a}$$

$$\hat{g}_{x+0.5,y} \equiv \mathrm{B}\left(g_{x+0.5,y}, p_{x+0.5,y}\right) \tag{3.8b}$$

$$G^x_{x,y} \leftarrow \alpha_{x+0.5,y}\hat{g}^x_{x+0.5,y} + (1-\alpha_{x+0.5,y})G^x_{x,y} \tag{3.8c}$$



$$g_{x,y+0.5} \equiv \left(\frac{G^x_{x-1,y} + G^x_{x,y} + G^x_{x-1,y+1} + G^x_{x,y+1}}{4} \; , \; G^y_{x,y} \right) \tag{3.9a}$$

$$\hat{g}_{x,y+0.5} \equiv \mathrm{B}\left(g_{x,y+0.5}, p_{x,y+0.5}\right) \tag{3.9b}$$

$$G^y_{x,y} \leftarrow \alpha_{x,y+0.5}\hat{g}^y_{x,y+0.5} + (1-\alpha_{x,y+0.5})G^y_{x,y} \tag{3.9c}$$

Figure 3.16: *Blending is applied at the location of desired x gradient samples* **(top)***, and the resulting x gradient is kept. Similarly, blending is applied at the location of desired y gradient samples* **(bottom)***. Note that both sets of intermediate $\hat{g}$ values are computed (in* (3.8b) *and* (3.9b)*) before either of the sets of desired gradients are updated (in* (3.8c) *and* (3.9c)*).*

Gradient Paint offers the user five choices of blending function B. I describe these choices below.

Additive blending sums the gradients from the paintbrush and background:

$$\mathrm{B}_{add}(g,p) \equiv g + p \tag{3.10}$$

It is useful when sketching, allowing one to build up lines over time, and for simple color and shadow adjustments. It is also useful when building up texture over multiple cloning passes.

Maximum blending selects the larger of the painted and background gradients:

$$\mathrm{B}_{max}(g,p) \equiv \begin{cases} g & \text{if } ||g|| > ||b|| \\ p & \text{otherwise} \end{cases} \tag{3.11}$$

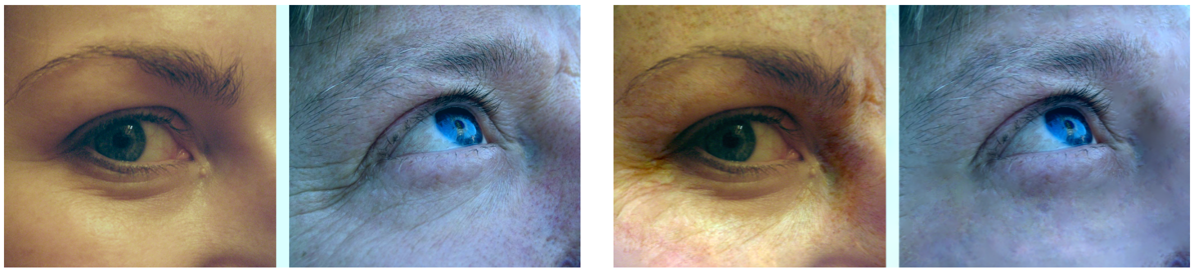


Figure 3.17: *Before* **(left pair)** *and after* **(right pair)** *swapping wrinkles. Crow's foot wrinkles were copied to the left face with the edge brush. Skin textures were exchanged using the clone tool. Painting with directional blending was used to enhance shading on the right face lost during the cloning.* Maximum *and* over *blending was used when copying wrinkles and skin texture;* minimum *blending was used to smooth the eye to the right; and* directional *and* additive *blending were used to adjust shading. Artist time: 15 minutes. (Original images by iamsalad,* left*, and burnt out Impurities,* right*, via flickr.)*

This blend is useful when cloning or copying edges, as it provides a primitive sort of automatic matting [Pérez et al. 2003]. My program also supports **minimum blending**, the logical opposite of maximum blending, which is occasionally useful when cloning smooth areas over noisy ones (e.g., removing wrinkles; see Figure 3.17).

Over blending simply replaces the background with the painted gradient:

$$\mathrm{B}_{over}(g,p) \equiv p \tag{3.12}$$

It is useful when cloning; or, with the gradient brush, to erase texture.

Directional blending, a new mode, enhances background gradients that point in the same direction as the brush gradients and suppresses gradients that point in the opposite direction:

$$\mathrm{B}_{dir}(g,p) \equiv \begin{cases} \left(1 + \frac{p \cdot g}{g \cdot g}\right) g & \text{if } g \neq 0 \text{ and } p \cdot g > -g \cdot g \\ 0 & \text{otherwise} \end{cases} \tag{3.13}$$

Directional blending is useful for lighting and contrast enhancement, as the artist may conceal her edits in existing edges, as in Figures 3.18 and 3.19.

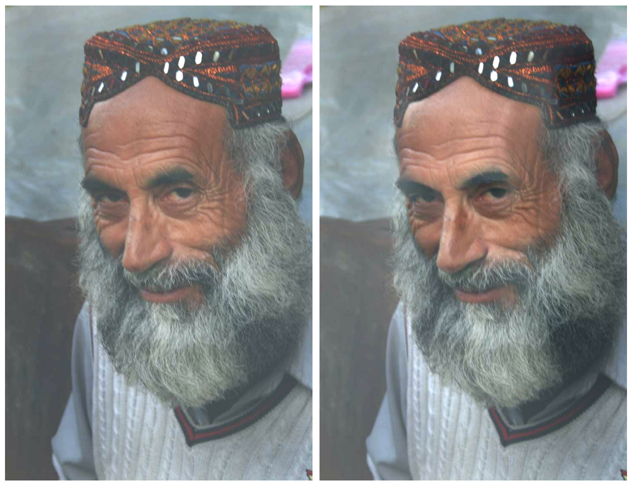


Figure 3.18: *Before* **(left)** *and after* **(right)** *contrast enhancement. The user applied the gradient brush with* directional *and* additive *blends. Strokes along the sides of the face enhance the shading, while strokes along the nose and eyebrows give the features more depth. Artist time: 3 minutes. (Original by babasteve via flickr.)*

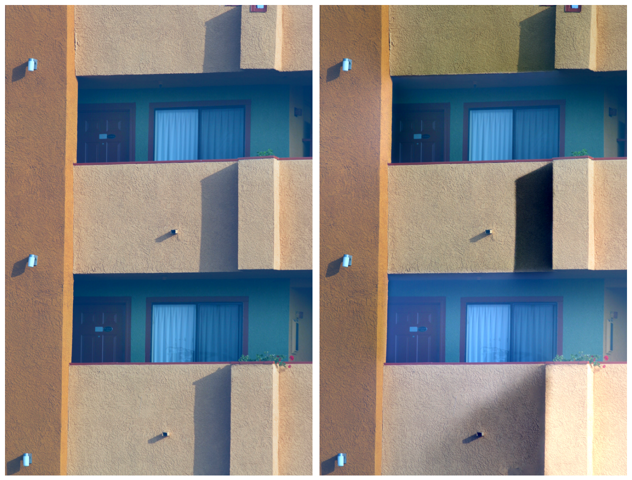


Figure 3.19: *Before* **(left)** *and after* **(right)** *shadow adjustments. The middle shadow was darkened using* directional *blending; the bottom shadow was removed by cloning over its top and left edges using* over *blending, then repainted as a more diffuse shadow using the gradient brush with* additive*; the top shadow remains unmodified, though the balcony has been tinted by applying colored strokes with* directional *blending to its edges. Artist time: 2.5 minutes. (Original by arbyreed via flickr.)*

3.3.4 Display

Since the desired gradients G^x, G^y may not exactly agree, my system tries to find an image u that has x and y differences that are close, in the least-squares sense, to these desired values (Figure 3.20).

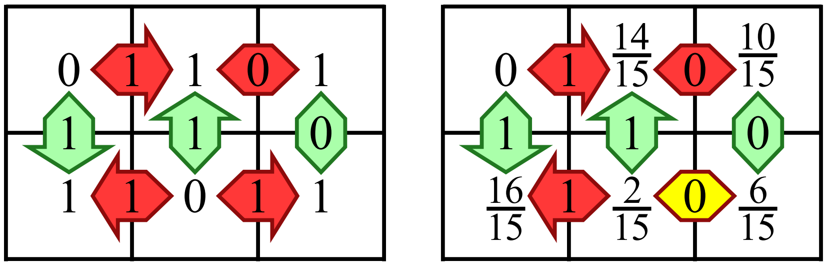


Figure 3.20: Left, *consistent desired gradients (green and red arrows) and a corresponding assignment of pixel intensities.* **Right**, *a gradient is changed (yellow highlight) to produce desired gradients that cannot be satisfied exactly. A least-squares fit for pixel intensities is shown. Notice how error is distributed across the image.*

Writing the portion of the least-squares objective involving a pixel $u_{x,y}$,

$$\begin{aligned}(3.14)\quad \operatorname*{argmax}_{u} \quad & \ldots + (u_{x+1,y} - u_{x,y} - G^x_{x,y})^2 + (u_{x,y} - u_{x-1,y} - G^x_{x-1,y})^2 \\ & + (u_{x,y+1} - u_{x,y} - G^y_{x,y})^2 + (u_{x,y} - u_{x,y-1} - G^x_{x,y-1})^2 + \ldots\end{aligned}$$

and setting the partial with respect to $u_{x,y}$ to zero, as one would when solving this convex least-squares system,

$$(3.15)\qquad 0 = 4u_{x,y} - u_{x+1,y} - u_{x-1,y} - u_{x,y+1} - u_{x,y-1} + G^x_{x,y} - G^x_{x-1,y} + G^y_{x,y} - G^y_{x,y-1}$$

one notices that this is simply the standard discretization of Poisson's equation:

$$(3.16)\qquad \nabla^2 u = f$$

Where f is computed from the edited gradients:

$$(3.17)\qquad f_{x,y} = G^x_{x,y} - G^x_{x-1,y} + G^y_{x,y} - G^y_{x,y-1}$$

This formulation of the display problem is a standard approach in graphics (e.g., [Pérez et al. 2003]); however, the solution methods used in previous methods are what limited them to interactive rather than real-time speeds.

My system solves this equation iteratively using the multigrid method. The multigrid technique was introduced by Fedorenko [1962], though – in the west – it was popularized by Brandt [1976] under the somewhat

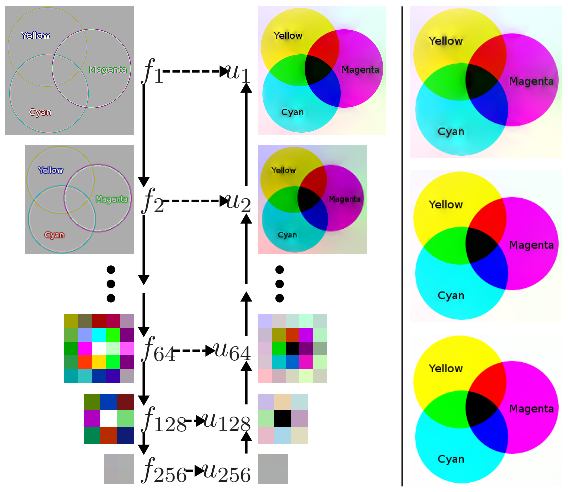


Figure 3.21: Left: *illustration of a single call to* `VCycle`*, with* f_1 *set to the gradient field of a 257×257 test image. Arrows show data flow.* **Right**: *The approximation is not perfect, but improves with subsequent iterations.*

unfortunate moniker MLAT (Muli-Level Adaptive Technique). For a general overview, the reader may refer to *Numerical Recipes in C* [Press et al. 1992], whose variable-naming conventions I follow. I describe my specific implementation in this section as an aid to those wishing to write similar solvers; my solver is a special case, customized for this specific domain and boundary conditions, of existing GPU multigrid solvers [Goodnight et al. 2003; Bolz et al. 2003].

In one iteration of my multigrid solver (in the jargon, a V-cycle with no pre-smoothing and two post-smoothing steps), I estimate the solution to the linear system $\mathcal{L}_h u_h = f_h$ by recursively estimating the solution to a coarser[4] version of the system $\mathcal{L}_{2h} u_{2h} = f_{2h}$, and then refining that solution using two Jacobi iterations. I provide an illustration in Figure 3.21, and give pseudo-code in Figure 3.22.

To effect these changes in grid resolution, the solver requires two operators $\mathcal{R}$ and $\mathcal{P}$. The restriction operator, $\mathcal{R}$, takes a finer f_h to a coarser f_{2h}. The interpolation operator, $\mathcal{P}$, expands a coarse u_{2h} to a finer u_h. My interpolation and restriction operators are both defined by the same stencil:

$$\mathcal{P} = \mathcal{R} = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix} \tag{3.18}$$

In other words, $\mathcal{P}$ inserts values and performs bilinear interpolation while $\mathcal{R}$ smooths via convolution with the above, then subsamples. The astute will notice that $\mathcal{R}$ is four times larger than the standard; this tends to keep f_h of a consistent magnitude – important, given the 16-bit floating point format of the target hardware.

[4]The subscript h denotes the spacing of grid points – so, u_{2h} contains one-quarter the pixels of u_h.

```
VCycle($f_h$):
    if (size($f_h$) == $1\times1$) return $[0]$
    $f_{2h} \leftarrow \mathcal{R} f_h$                              ;Restrict $f$ to coarser grid
    $u_{2h} \leftarrow \texttt{VCycle}(f_{2h})$                     ;Approximate coarse solution
    $u_h \leftarrow \mathcal{P} u_{2h}$                              ;Interpolate coarse solution
    $u_h \leftarrow \texttt{Relax}(u_h, f_h, x_{h0})$               ;Refine solution
    $u_h \leftarrow \texttt{Relax}(u_h, f_h, x_{h1})$               ;Refine solution (again)
    return $u_h$
Relax($u_h$, $f_h$, $x$):
    return $\frac{1}{m_h - x}\left(f_h - (\mathcal{L}_h - (m_h - x)I)u_h\right)$    ;Jacobi
```

Figure 3.22: *Pseudo-code implementing my multigrid-based integration algorithm. Each frame,* `VCycle` *is called on the residual error, $f_1 - \mathcal{L}_1 u_1$, to estimate a correction to the current solution. The variable m_h refers to the m associated with $\mathcal{L}_h$ (Equation 3.19). I set $x_{h0} = -2.1532 + 1.5070 \cdot h^{-1} + 0.5882 \cdot h^{-2}$ and $x_{h1} = 0.1138 + 0.9529 \cdot h^{-1} + 1.5065 \cdot h^{-2}$.*

The operator $\mathcal{L}_h$ the algorithm solves for at spacing h is given by a more complicated stencil:

(3.19)
$$\mathcal{L}_h = \begin{bmatrix} c & e & c \\ e & m & e \\ c & e & c \end{bmatrix}, \text{ with } \begin{bmatrix} m \\ e \\ c \end{bmatrix} = \frac{1}{3h^2} \begin{bmatrix} -8h^2 - 4 \\ h^2 + 2 \\ h^2 - 1 \end{bmatrix}$$

Notice $\mathcal{L}_1 = \nabla^2$. These coefficients have been constructed so that the solution at each level, if linearly interpolated to the full image size, would be close as possible, in the least-squares sense, to the target gradients. This choice is consistent with Bolz et al. [2003], though they do not provide an explicit formula and use a different justification.

In my GPU implementation I store all data matrices as 16-bit floating-point textures, and integrate the three color channels in parallel. I use 0-derivative (i.e., Neumann with value zero) boundary conditions, since these seem more natural for editing; however, this implies that the solution is only defined up to an additive constant. The solver resolves this ambiguity by white-balancing the image. As interactive and consistent performance is important for my application, I run one `VCycle` every frame instead of imposing a termination criterion.

3.4 Evaluation

I evaluated my integrator on gradient fields taken from images of various sizes and aspect ratios. My test set included a high-resolution full scan of the "Lenna" image and 24 creative-commons licensed images drawn from the "recently added" list on flickr. To characterize how quickly my integrator converges to the right solution, I modeled a worst-case scenario by setting u to a random initial guess, then calculated the root-mean-square of the residual, $\nabla^2 u - f$, and of the difference between the x-gradient of u and G^x; both are plotted in Figure 3.23 (left). In practice, images are recognizable after the first integration step (i.e., call to `VCycle`) and nearly perfect after the second – Figure 3.23 (right).

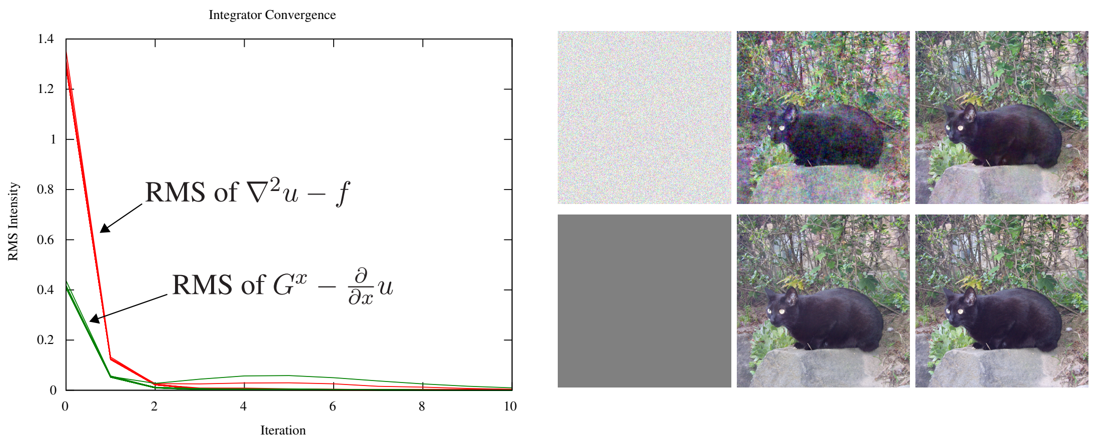


Figure 3.23: **Left:** *Root-mean-square (over all pixels) error after a given number of V-cycles. Results for all test images are overplotted. Image intensities are represented in the* $[0, 1]$ *range. The increase in error around the fourth iteration is due to approximations made when handling boundary conditions of certain image sizes.* **Right:** *Integrator convergence on an example image, starting from two different initial guesses. Images, from left to right, show the initial condition, the result of one V-cycle, and the result of two V-cycles.*

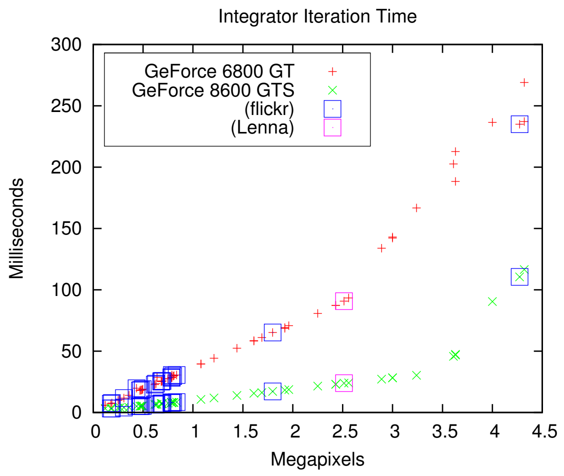


Figure 3.24: *Milliseconds per integrator cycle; performance is linear until data exceeds video memory – about 2.5 megapixels on modest hardware, 3.5 on the newer hardware. Un-boxed points are synthetic examples.*

I also performed an eigenvalue analysis of a 65×65 problem, similar to that used by Roberts to analyze his method [2001]. I achieve a convergence rate, $\bar{p}_0 = 0.34$, indicating that the algorithm removes more than 65% of the remaining error each iteration. This is on par with Roberts' reported value for conventional multigrid. The constants x_{h0} and x_{h1} appearing in my algorithm (Figure 3.22) were selected by numerical optimization of this $\bar{p}_0$.

Timing for a single integrator cycle (the bulk of the work the system does each frame) is recorded in Figure 3.24. Because integration time is not content-dependent, I augmented my test set with 37 blank images to check integration speeds over a wider range of sizes and aspect ratios. The two computers used for testing were a several-year-old desktop with an Athlon64 3400 CPU, 1 GB of RAM, and a GeForce 6800 GT GPU, running Gentoo Linux – the sort of system that many users have at their desks today – and a moderately priced ($1500) workstation with a Core2 Quad Q6600 CPU, 4 GB of RAM, and a GeForce 8600 GTS GPU, running Windows Vista. Even on the more modest hardware, my software comfortably edits 1-megapixel images at around 20fps. Algorithmic scaling is theoretically linear in the number of pixels, and this tendency is very pronounced as long as all textures are resident in video memory.

3.5 Other Applications

Because my integrator is so fast, it might be useful for real-time gradient-domain compositing of video streams – for instance, of a computer-generated object into a real scene for augmented reality, or of a blue-screened actor into a virtual set. Care would have to be taken with frame-to-frame brightness and color continuity, both of the scene and of the pasted object, either by adding a data term to the integration or by introducing more complex boundary conditions.

With a DSP implementation, my integrator could serve as the core of a viewfinder for a gradient camera [Tumblin et al. 2005]. A gradient camera measures the gradients of a light field instead of its intensity, allowing HDR capture and improving noise resilience. However, one drawback is the lack of a live image preview ability. Some computational savings would come from only having to compute a final image at a coarser level, say, a 256×256 preview of a 1024×1024 image. My choice of $\mathcal{L}_h$ and $\mathcal{R}$ guarantees that this coarser version of the image is still a good approximation of the gradients in a least-squares sense.

This method may also be applicable to terrain height-field editing. Users could specify where they wanted cliffs and gullies instead of having to specify explicit heights.

3.6 Discussion

With a real-time integrator, gradient editing can be brought from the realm of edit-and-wait filters into the realm of directly controlled brushes. My simple multigrid algorithm, implemented on the GPU, can handle 1-megapixel images at 20fps on a modest computer. My editor allows users to paint with three brushes: a simple gradient brush useful for sketching and editing shading and shadows, an edge brush specifically designed to capture and replay edges from the image, and a clone brush. Each brush may be used with different blending modes, including directional blending mode which “hides” edits in already existing edges.

Of course, this system is not the only way one might go about painting in the gradient domain; different integration schemes and new brushes are both possible areas for future work.

3.6.1 Integrator

My multigrid integrator, while adequate to what is currently required of it, is not without downsides. More memory is required than in a conventional image editor; G^x and G^y must be stored (though any integrator will need to deal with that), as well as multi-scale versions of both f and u, not to mention temporary space for computations – this requires about 5.6 times more memory than just storing u; for good performance, this memory must be GPU-local. Conventional image editors handle large images by tiling them into smaller pieces, then only loading those tiles involved in the current operation into memory. Such an approach could also work with my integrator, as all memory operations in this algorithm are local. Alternatively, I might be able to minimize main-memory to GPU transfers by adapting the streaming multigrid presented by Kazhdan and Hoppe [2008].

Another option that presents itself is to avoid storing G^x and G^y altogether in favor of f. This "Laplacian-domain image editing" would have a smaller memory footprint; however, the sum-of-second-derivatives may be harder for users to reason about than edges. Also, the various blending modes would be trickier to define in this domain.

One of the main speed-ups of my integrator is that its primitive operations map well onto graphics hardware. A Matlab CPU implementation ran only a single iteration on a 1024×1024 image in the time it took for a direct FFT-based method (see Agrawal and Raskar [2007]) to produce an exact solution (about one second). Thus, for general CPU use I recommend using such a direct solution. This result also suggests that GPU-based FFTs may be worth future investigation; presently, library support for such methods on older graphics cards is poor.

My integrator is not suitable for problems that have value as well as gradient constraints. Similarly, state-of-the-art real-time solvers that handle Poisson interpolation of value constraints (e.g., [Orzan et al. 2008; Jeschke et al. 2009]) do not adapt readily to allow gradient constraints. A real-time solver that could handle both sorts of constraints would allow for interesting hybrid techniques. One possible starting point for such a solver would be a combinatorial multigrid – an advanced multigrid approach proposed by Koutis [2007]. However, while it is theoretically very fast, the somewhat complicated partitioning employed by this method could result in less than real-time performance.

3.6.2 Paint Tools

While I present three brushes, this is unlikely to exhaust the possibility space of gradient domain paint tools. For instance, one can imagine tools that operate on areas (analogous to intensity domain flood-fill), or brushes that adapt their output dynamically to (e.g.) enforce that a certain pixel remains a certain color. I suspect there are more interesting gradient-domain tools waiting to be discovered.

Additionally, there is room for refinement in the brushes that I present herein. The looping of the edge brush could be made more intelligent, either using a video-textures-like approach [Schödl et al. 2000], or by performing a search to find good cut points given the specific properties of the current stroke. Edge selection might be easier if the user were provided with control over the frequency content of the copied edge (e.g., to separate a soft shadow edge from high-frequency grass). And tools to limit the effect of edits, through either modified boundary conditions or a data term in the integrator, would be convenient. Like any image editing system, my paint tools can produce artifacts – one can over-smooth an image, or introduce dynamic range differences or odd textures. However, also like any image editing system, these can be edited-out again or the offending strokes undone.

I have informally tested my gradient-domain paint system with a number of users, including computer graphics students and faculty, art students, and experienced artists. Users generally had trouble with three aspects of my system:

- **Stroke Directionality** – artists are accustomed to drawing strokes in whatever direction feels natural, so strokes in different directions having different meanings was a stumbling block at first.
- **Relative Color** – both technical users and artists are used to specifying color in an absolute way. Users reported that it was hard to intuit what effect a color choice would have, as edges implicitly specify both a color (on one side) and its complement (on the other).
- **Dynamic Range** – it is easy to build up large range differences by drawing many parallel strokes. Users were often caught by this in an early drawing.

The interactivity of the system, however, largely mitigated these concerns – users could simply try something and, if it didn't look good, undo it. Additionally, some simple interface changes may help to address these problems; namely: automatic direction determination for strokes, or at least a direction-switch key; a dual-eyedropper color chooser to make the relative nature of color more accessible; and an automatic range-compression/adjustment tool to recover from high-dynamic range problems.

In the future, I would like to see gradient-domain brushes alongside classic intensity brushes in image editing applications. The Gradient Paint system takes an important step toward this goal, demonstrating an easy-to-implement real-time integration system coupled to an interface with different brush types and blending modes.

Gradient Paint demonstrates that by taking a perception-motivated image feature (edges) and building a real-time editing framework, one can create a powerful new image editor which makes many tasks, like the adjustment of shadows and shading, much easier to perform.

Chapter 4

Local Layering

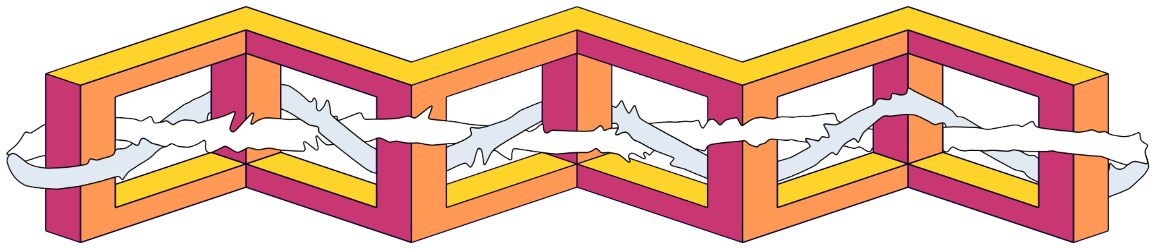


Figure 4.1: *Local layering allows complicated overlapping between graphical objects without manually splitting layers or painting masks.*

Traditional compositing programs require that layers appear in the same order everywhere they overlap. In this project, I relax this strict ordering requirement. In keeping with my thesis, I do so in a way that introduces an intuitive local feature (stacking order within regions of overlap) and preserves ambiguity (stacking edits propagate only as far as required to maintain consistency).

Specifically, I introduce a method of storing local stackings – the *list-graph* – and two layer rearrangement primitives `Flip-Up` and `Flip-Down` that operate on it. These primitives allow artists to place one layer above or below another layer *locally*. This local rearrangement is intuitive because it mirrors what would happen if one were to attempt to adjust the stacking of paper cut-outs – local edits must propagate a to a small neighborhood to prevent the paper from tearing.

These primitives allow me to replace the conventional, global-order-adjusting, *layers dialog* with a new *local layers dialog* (Figure 4.2). With this local ordering control, users are able to create layers that weave over, under, and through one another (Figures 4.1, 4.3), mimicking the behavior of objects in the real world (Figure 4.4). The resulting system can even be applied to animations.

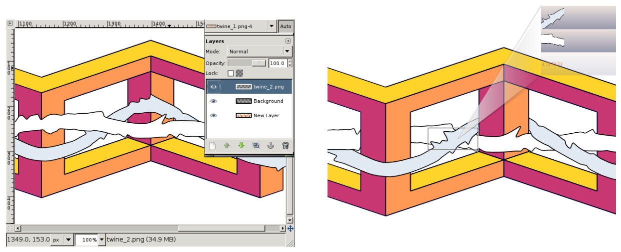


Figure 4.2: Left: *A standard paint program (GIMP pictured) provides only a global ordering of layers.* **Right:** *My local layering prototype allows a – possibly different – ordering at each region of overlap. Here the local layering dialog is being used to control the overlap of the white and gray ropes at one of their crossing points.*

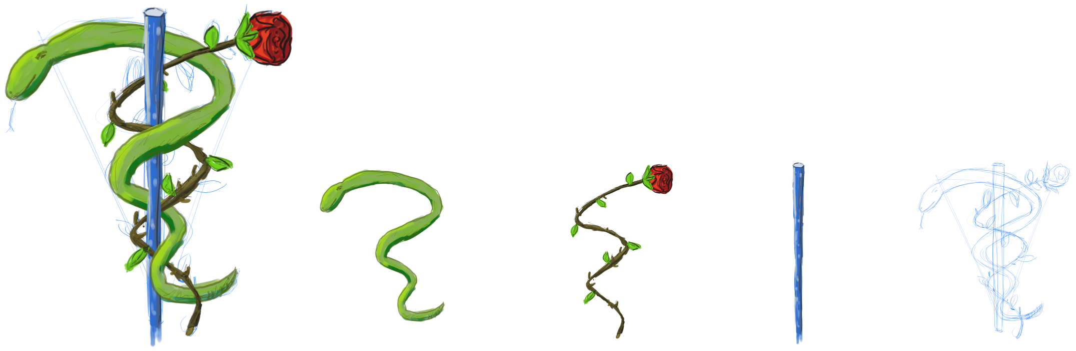


Figure 4.3: *Complex overlapping creates visual interest; this composition uses only four layers – the snake, the rose, the staff, and the background.*

4.1 Contribution

The main contribution of this work is a general representation for images with local stacking (the list-graph), rearrangement operators (`Flip-Up`, `Flip-Down`) which correspond to the expected layer ordering operations, and proofs that these operators are correct and sufficient to reach all legal orders. Additionally, an ad hoc temporal coherence scheme is given which enables editing of layer contents. This has the interesting side effect of allowing one to produce real-time-viewable impossible figures.

This approach and these operators are demonstrated to work on raster images, with stacking local to overlap regions, and raster animations, with stacking local to spatio-temporal overlap regions. Operation on vector images and animations should be similar, given accurate and stable intersection operators (e.g., a planar map implementation).

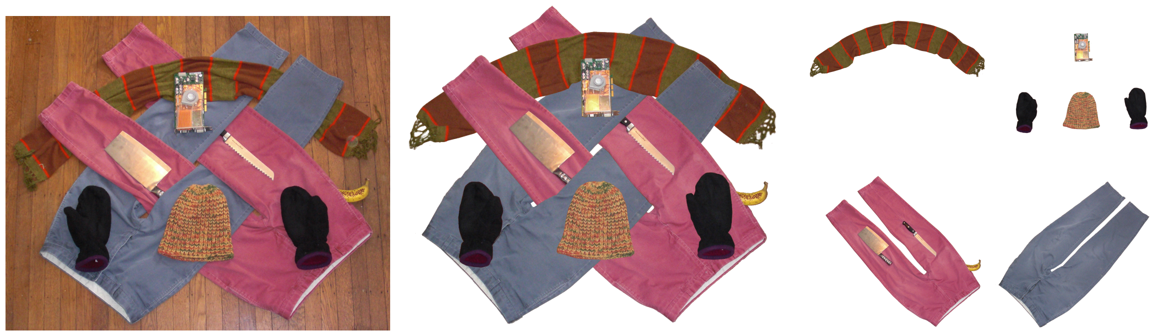


Figure 4.4: **Left,** *objects stacked in the real world.* **Middle,** *objects stacked with local layering.* **Right,** *layers used.*

4.2 History

In graphics, the notion of representing scenes as stacks of partially transparent layers was largely borrowed from the film industry practice of optical compositing. Optical compositing is the process of combining elements from different scenes onto one film. The simplest kind of optical compositing is in-camera multiple exposure, illustrated in Figure 4.5; here, various takes are photographed onto the same reel of film, rewinding between takes. This accumulates multiple images into the same negative.

However, multiple-exposure compositing breaks down when objects overlap (Figure 4.6). This is because light that should be occluded by an object is not. This problem can be solved with the use of special extra films called "masks," which are stacked with other films to block unwanted light. The most rudimentary way to perform masking is by shooting the foreground subject against a very dark background and then exposing the result onto high-contrast film. This process is illustrated in Figure 4.7. Though mask-based methods were already in use in the 1910s, they remain the state-of-the-art in optical compositing – albeit with ever more sophisticated ways of actually pulling masks.

In the 1970s, graphics firms began to experiment with duplicating this process entirely on the computer, leading to innovations like premultiplied alpha and the compositing algebra of Porter and Duff [1984]. However, the main contribution of graphics to compositing was the subtle shift from thinking of transparency as something that lived on a separate mask film to thinking of it as a part of each pixel, like color. It is this notion ("intrinsic alpha") which encourages us to think of pixels in raster images as belonging to discrete objects instead of image-sized layers with associated image-sized masks. More information about the history of compositing is available in a technical memo by Alvy Ray Smith [1995], and makes for interesting reading. Compositing remains largely unchanged in today's image- and video-editing programs (e.g., [Kimball and Mattis 1995–2009; Apple 1999]), though, much like in film, graphics researchers have devised increasingly devious ways of constructing masks.

The inverse problem, that of creating a stack of layers corresponding to an image, is of interest to vision researchers as a primitive for image understanding [Nitzberg and Mumford 1990]. In real scenes, layers may be segmented using any number of features (e.g. motion [Adelson and Wang 1994]). If all object contours are known, such a stack of layers – corresponding to front- and back-facing regions of the object – may

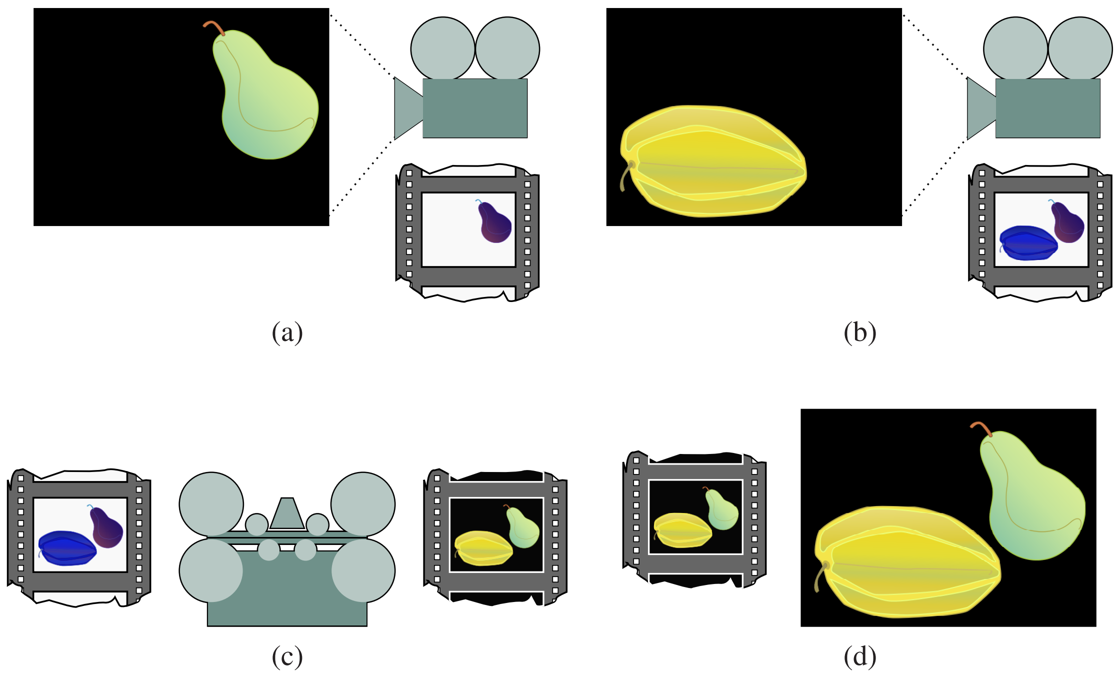
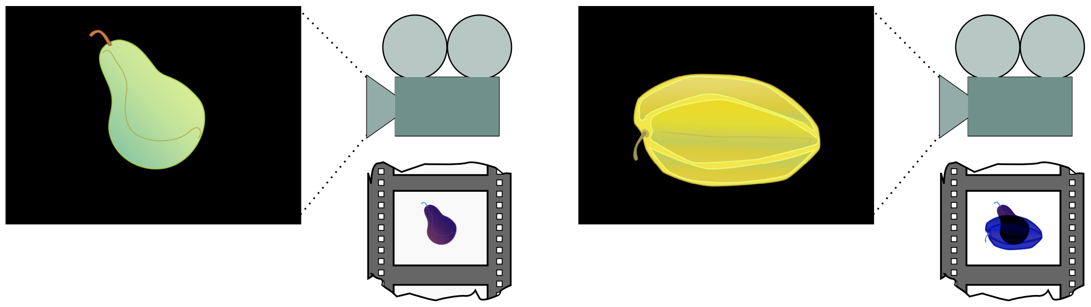


Figure 4.5: *In-camera compositing with multiple exposures.* **(a)** *The first scene element is shot.* **(b)** *A second element is photographed onto the same negative.* **(c)** *The produced negative is printed.* **(d)** *The final image contains both elements.*

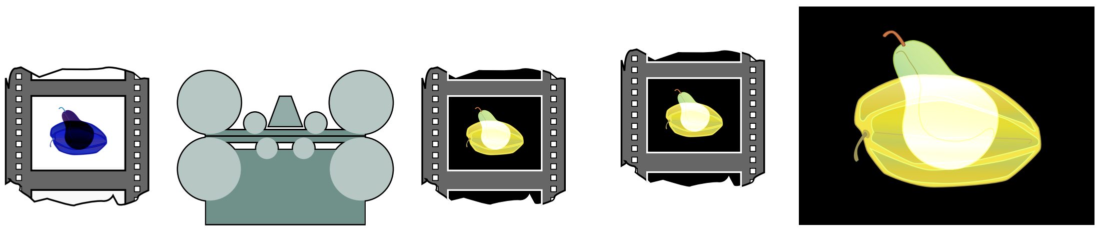
be constructed using the paneling algorithm of Willams [1997]. This approach has been systematized for modeling [Karpenko and Hughes 2006; Cordier and Seo 2007].

When combining images of separate objects in a 3D scene, simple global layering is not enough, due to occlusion cycles. If available, depth information can be used to provide a local stacking [Duff 1985]. Alternatively, layers can be modified by removing occluded pieces [Snyder and Lengyel 1998]. The core of this approach – compiling an occlusion graph into a series of compositing operations – might be useful to accelerate display of the stackings constructed by my method. However, because Snyder and Lengyel rely on whole-image operations, their technique cannot handle the situation where two layers A and B must be composited A over B in one place and B over A in another. Stacking results similar to those my method produces may be attained manually by reproducing such a process – either designing 3D geometry for a 2D scene (e.g., [Debevec et al. 1996]) or by painting out parts of objects that are occluded – though to do so is tedious.

My construction works on the adjacency graph of regions of overlap of graphical objects. This is similar, in spirit, to a planar map [Baudelaire and Gangnet 1989] – a method of representing a vector drawing in which all curves are treated as lying in the same plane and separate fills and strokes may be assigned to any region or portion of an edge. Just as users can simulate local layering in a stacked representation by duplicating layers and painting masks, users can simulate local layering (and stacking) in a planar map by adjusting fills and colors of regions, potentially touching many adjacent regions to make one visibility edit. In contrast,



(a) (b)



(c) (d)

Figure 4.6: *In-camera compositing fails when subjects overlap.* **(a)** *The first scene element is shot.* **(b)** *A second element is photographed onto the same negative, over-exposing the overlapped area.* **(c)** *The produced negative is printed.* **(d)** *The final image contains both elements added together without occlusion.*

my system maintains structural information about objects and stacking in each overlap region, and thus can automatically re-stack adjacent regions when visibility editing and compute proper pixel colors in overlaps of transparent objects.

Where my method relaxes the rigid structure of a layer-based approach, Asente et al. [2007] propose a method that infers structure from a planar map in order to allow more intuitive editing operations. As part of their system, they estimate object colors for closed curves and extract a potential stacking order by examining their regions of overlap. This allows them to propagate region fill colors in a reasonable way as curves are moved. As these stacking orders are estimates based on region coloring, they cannot reliably be made in drawings with, e.g., artist-simulated transparency. This is acceptable as it is not the goal of Asente et al. to calculate the stacking of objects; rather, the inferred stacking is one of many cues used to determine reasonable behavior during edits. In contrast, my system explicitly maintains a consistent stacking order and allows it to be edited directly.

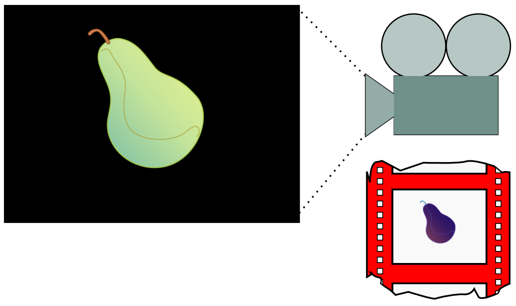
Wiley [2006] presents a vector-graphics drawing system, Druid, that represents regions and their stacking as valid over/under labellings of the intersections of a set of boundary curves. This system is able to elegantly handle self-overlapping surfaces, and (in contrast to planar maps) does produce a full visibility-order for each region of overlap. While Druid presently only supports solid fill colors, one could imagine extending it to produce results similar to those herein presented by constructing closed curves around layer boundaries of raster objects.

Though my proposed algorithm does not handle self-overlaps, it does have several technical advantages over that of Wiley: My program's core re-stacking operator is polynomial time, whereas the core re-labelling method of Druid involves a worst-case exponential-time search (though heuristics and construction of equivalence classes make the common case scale reasonably well). Rendering from my representation is fast, because the stacking order of surfaces inside regions is known; Druid must infer relative surface depths when rendering, which is time-consuming. The editing operations I present are sufficient to reach all valid orderings; such a result would be difficult to prove given Wiley's editing operations (though it is likely true). Finally, it is not clear how to generalize the edge-crossing representation in Druid to animation, whereas my adjacent-region method was easy to generalize by giving regions temporal extent.

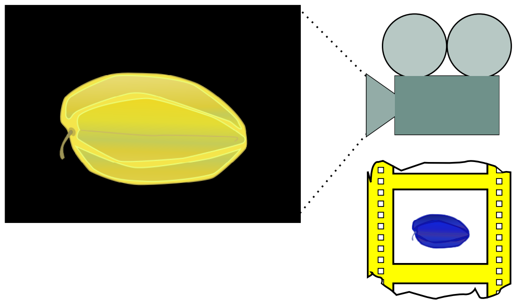
From a practical standpoint, my system is simple to describe, prove properties of, and implement. Also, because I extend the notion of a conventional layers dialog, my stacking adjustment method may be more familiar to users.

Igarashi et al. [2005] give a heuristic method, based on continuously monitored and updated depth values, to generate a temporally coherent stacking order for a deformable 2D object. Their method is designed for self-occlusions of a single object and does not make guarantees about an intersection-free result, while my method is designed for complex layering of multiple objects and is provably correct.

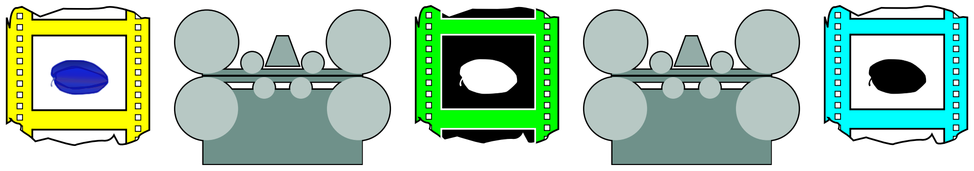
Limited local layer re-ordering support is provided by the Push-Back tool in Real-Draw PRO [Mediachance 2001–2009]. This tool moves the top layer of a stack to some other point in the stack in a user-selected region. This automates the traditional mask-based method of achieving local re-ordering, but retains the drawback of requiring the user to manually define the region of effect. In contrast, my method can locally re-order all layers and automatically extends layer edits as far as is needed to prevent layer interpenetration.



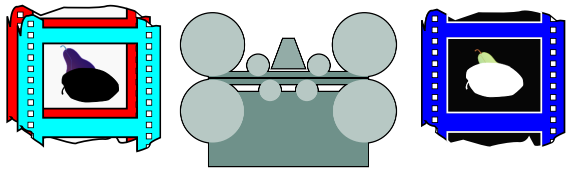
(a)



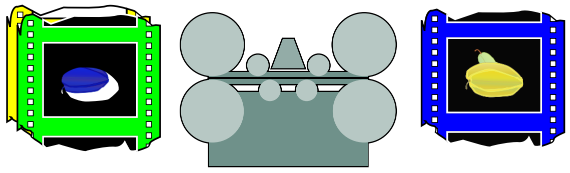
(b)



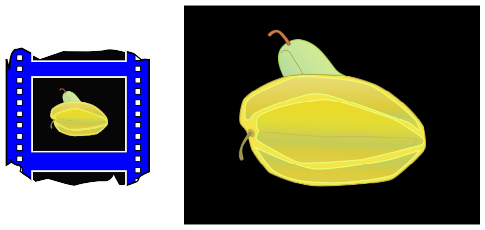
(c)



(e)



(f)



(g)

Figure 4.7: *Using a mask to resolve object overlap in an optical composite.* **(a,b)** *The scene elements are photographed onto different negatives.* **(c)** *The foreground object is exposed onto high-contrast film to create a background mask; this is exposed again to create an foreground mask.* **(d)** *The background element and the background mask are stacked and exposed onto the final film. The background mask prevents the part of the film where the foreground element will occur from being exposed.* **(e)** *The foreground element and mask are stacked and exposed onto the final film. The mask ensures that the background is not doubly exposed.* **(f)** *The final composite contains properly overlapping foreground and background elements. In a production environment, this composite would need to be printed to a negative again for film duplication.*

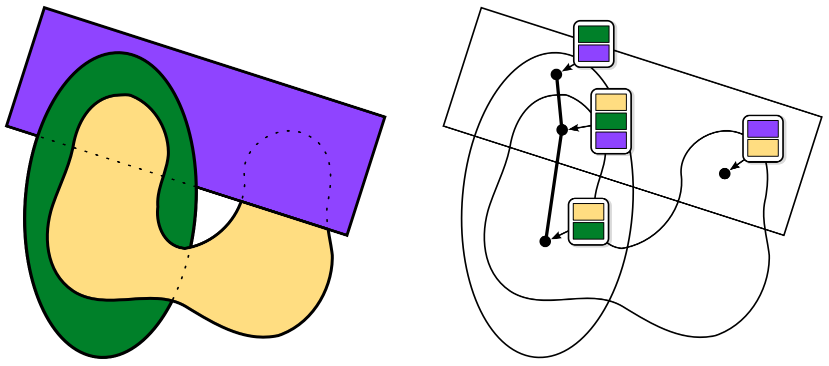


Figure 4.8: Left, *an example figure containing three objects. Dotted lines indicate occluded contours.* **Right,** *the corresponding regions of overlap and list-graph. Lists containing one or fewer elements are not shown.*

4.3 Method

With local layering, layers can be ordered just as one would order paper cut-outs, weaving and overlapping but never passing through one another. In this section, I examine the techniques I use to achieve this goal. Specifically, I discuss how local stacking is represented; the operators invoked when the user instructs the program to change the ordering of layers in a region of overlap; and the ability of these operators to navigate all possible stackings. Additionally, I talk about how my program handles the case where layers change content (e.g., if the user drags or paints).

4.3.1 Data structure

Where a global layering approach needs only one list to track the relative ordering of objects, my approach requires lists for each region of overlap. To store these local orderings my program uses a structure called a list-graph. A drawing and the corresponding list-graph appear in Figure 4.8.

Definition 4.3.1 (List-graph)**.** A *list-graph* G is an undirected graph with a list stored at each vertex. I write $L \in G$ to indicate that list L appears at some vertex of G, and $x < y \in L$ to indicate object x appears under – though not necessarily directly under – object y in the stacking given by list L. It is not the case that all objects must appear in all lists, or that they must appear in the same order in those lists which they inhabit.

To construct a list-graph from a set of overlapping objects, one first partitions the image into regions of overlap – that is, connected regions of the plane covered by the same set of objects. For each region of overlap, a list-graph vertex is created that contains a stacking order for the layers that appear in that region. Edges are created between any two vertices whose associated regions of overlap are adjacent in the image. The stacking order stored at each vertex may either be initialized to a global order or set based on an existing ordering (e.g., when moving a layer), which I will demonstrate later.

Since my operators are defined in terms of areas of overlap and the list-graph abstraction, they apply equally

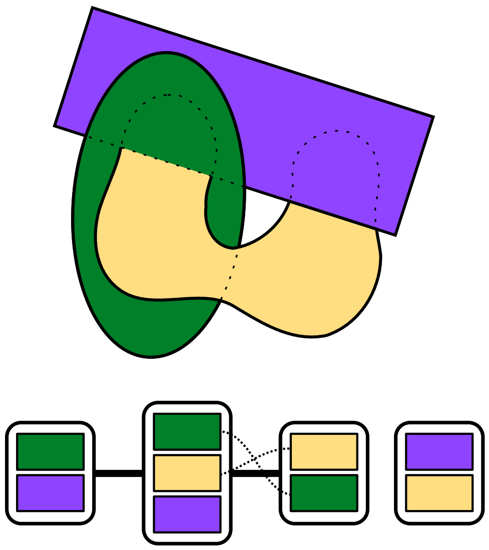


Figure 4.9: Above, *a version of the stacking given in Figure 4.8 containing an undesirable intersection between the green oval and tan blob. The list-graph,* **below**, *is inconsistent (inversion indicated with dotted lines). Lists containing one or fewer elements are not shown.*

well in a raster or vector setting (as long as the primitive objects contain no self-overlap). However, my prototype uses raster graphics exclusively, and I consider a layer to exist anywhere it has non-zero alpha. More implementation details are given in Section 4.5.

It is important that layers do not pass through each other in the final image. I formalize this notion as consistency:

Definition 4.3.2 (Consistency)**.** A list-graph G is *consistent* if for all adjacent lists $A, B \in G$ and all pairs of layers $x, y \in A \cap B$, $x < y \in A \iff x < y \in B$.

An example of the sort of artifact produced by an inconsistent list-graph is given in Figure 4.9.

4.3.2 Local Order Adjustment

Of course, being able to represent different orderings is of no use if users have no way of specifying them. In order to make local ordering adjustments, I introduce the `Flip-Up` and `Flip-Down` operators (pseudo-code for `Flip-Up` appears in Figure 4.10). These operators are local versions of the "move above" and "move below" operators in a global layering setting. When `Flip-Up`(L, x, t) is called, the procedure first makes sure that $x, t \in L$, then proceeds to slowly inch x up in the local stacking – calling `Flip-Up` on all the adjacent lists after each increase to make the list-graph consistent. (If it were, instead, to jump x immediately above t then the information about x's new order with respect to any element $x < b < t$ would not be propagated.) `Flip-Down` proceeds similarly, lowering x until it is below t.

However, while the rearrangement operators in a global setting are trivial, these local operators are not, and

```
0: Flip-Up($L, x, t$):
1:     **if** ($x \notin L \vee t \notin L$) **return**
2:     **while** ($x < t \in L$):
3:         Let $y$ be the element directly above $x$ in $L$.
4:         Swap $x$ and $y$ in $L$.
5:         **for** ($L'$ adjacent to $L$):
6:             Flip-Up($L', x, y$)
```

Figure 4.10: *Pseudo-code implementing the* `Flip-Up` *operator, which places layer x above layer t in list L while maintaining list consistency.* `Flip-Down` *is defined similarly, replacing the condition in line 2 with $x > t \in L$ and the word "above" with "below".*

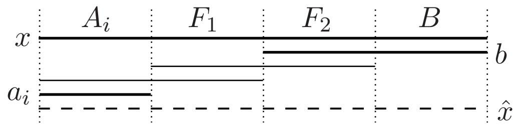


Figure 4.11: *Side view of scenario in Lemma 4.3.5; if it was the case that $x < a_i \in A_i$ before a call to* `Flip-Up`*, then it must have been the case that $x < b \in B$ as well, i.e., x was at $\hat{x}$.*

thus need to be proven correct.

Theorem 4.3.3 (Termination of `Flip-Up`)**.** *On list-graph G,* `Flip-Up`*(L, x, t) terminates in $O(\#edges \cdot \#layers)$ time.*

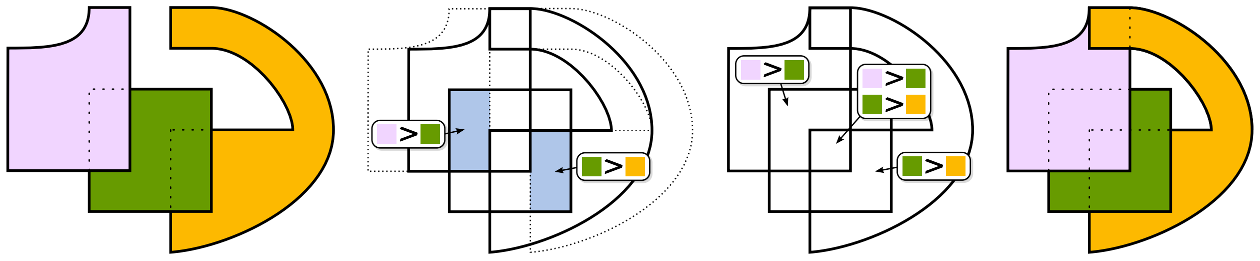
Proof: If `Flip-Up` is called on a list L with no neighbors, then $O(\#\text{layers})$ work is needed. Otherwise, with the proper data structures, $O(1)$ work is done per execution of the recursive call on line 6. I charge this work to the edge traversed by the recursive call. Consider an edge between lists A and B. Since x must be moved up one step in either A or B (line 4) to traverse the edge between them, and x is never lowered, at most $|A| + |B| = O(\#\text{layers})$ calls are made. Multiplying, at most $O(\#\text{edges} \cdot \#\text{layers})$ work was done. ■

Theorem 4.3.4 (Soundness of `Flip-Up`)**.** *After* `Flip-Up`*(L, x, t) runs on a consistent list-graph G, the list-graph remains consistent and x appears above t in list L.*

Proof: Notice that `Flip-Up` will not terminate without $x > t \in L$, so all I need to show is that G remains consistent. Proceed by contradiction. Assume that there is a pair of elements, shared by adjacent lists A, B, which appear in different orders in each list. Since `Flip-Up` never changes the relative order of non-x elements, one of these must be x. Call the other y. Without loss of generality, let $x < y \in A$ and $x > y \in B$. Since `Flip-Up` never moves x down, it must be the case that $x > y \in B$ because of the action of `Flip-Up`. But this leads to a contradiction, because when `Flip-Up` placed x above y in B it would have recursed to A and placed x above y there as well. ■

The termination and soundness of `Flip-Down` proceed similarly.

Now I need to address a more subtle point. While I have shown that my operators always take consistent list-graphs to consistent list-graphs, it could be the case that there are some consistent configurations that



(a) (b) (c) (d)

Figure 4.12: *Maintaining a temporally coherent stacking.* **(a)** *The original stacking.* **(b)** *The updated layer positions, with local votes cast in the image-space overlap between the old stacking regions (dotted) and the new (solid).* **(c)** *After vote propagation.* **(d)** *Final stacking; all votes satisfied. The overlap at the top was not constrained, so was chosen based on a given tie-breaking order (here prioritizing the orange crescent).*

cannot be reached with `Flip-Up` and `Flip-Down`; this would – no doubt – be infuriating to users. First, though, I need a lemma:

Lemma 4.3.5 (Invertability)**.** *The action of* `Flip-Up` *may be inverted by a sufficient number of calls to* `Flip-Down`.

Proof: I show that the results of a call to `Flip-Up`(L, x, t) may be undone by the following procedure: At step i, find a remaining inversion – that is, choose layer a_i in list A_i such that it was the case that $x < a_i \in A_i$ before the call to `Flip-Up`, and it is currently the case that $x > a_i \in A_i$. Call `Flip-Down`(A_i, x, a_i).

To show this approach is correct, I demonstrate that each call to `Flip-Down` strictly decreases the set of remaining inversions. Specifically, consider $b \in B$ such that $x > b \in B$ before the ith call to `Flip-Down` and $x < b \in B$ after the call. (One possible scenario is illustrated in Figure 4.11.)

Before the call to `Flip-Down`(A_i, x, a_i), there must have been adjacent lists $A_i, F_1, \ldots, B$ such that

$$a_i \leq f_1 < x \in A_i, f_1 \leq f_2 < x \in F_1, \ldots, f_k \leq b < x \in B$$

(This is simply writing down the condition for `Flip-Down` to have recursed to $b \in B$.) These inequalities must have also been true just after `Flip-Up` returned, since the intervening $i-1$ calls to `Flip-Down` can't have raised x.

Consider the same adjacent lists before the call to `Flip-Up`. By hypothesis, $x < a_i \in A_i$; so, by consistency:

$$x < a_i \leq f_1 \in A_i \Rightarrow x < f_1 \leq f_2 \in F_1 \Rightarrow \ldots \Rightarrow x < f_k \leq b \in B$$

Thus, the ith call to `Flip-Down` has returned b and x to their original order; the set of inversions strictly decreases; and I am done. ■

With that lemma in hand, we dive into the main theorem:

Theorem 4.3.6 (Sufficiency)**.** *The* `Flip-Up` *and* `Flip-Down` *operators allow any consistent configuration of given list-graph G to be reached from any consistent starting position.*

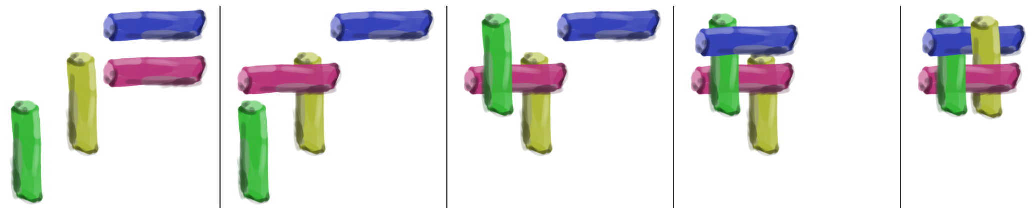


Figure 4.13: *The user weaves together four layers by dragging them. This example relies on temporal coherence to maintain stacking information. My prototype places the active layer on top in new overlaps.*

Proof: First, I show how to take any consistent configuration of G to a canonical form where the layers $1, \ldots, n$ appear bottom to top in numerical order:

Proceed by induction on current layer i. Assume layers $i+1, \ldots, n$ appear in numerical order at the top of any $L \in G$ in which they occur. Now, call `Flip-Up`(L, i, y) for all $L \in G$ and $1 \leq y < i$. This moves i to the top of every $L \in G$ in which it appears, just before elements $i+1, \ldots, n$. `Flip-Up`(L, i, k) will never be called with $k > i$, because that would imply that $k < i \in L$, which contradicts the assumption that all $k > i$ are at the top of L. Therefore one can place the layers in a canonical order with calls to `Flip-Up`.

Of course, this means that – by Lemma 4.3.5 – one can take the canonical form to any consistent form G' using calls to `Flip-Down`. ■

In practice, using `Flip-Up` and `Flip-Down` to navigate through the space of consistent stackings is quite intuitive, despite the cumbersome construction used in the proof above.

4.3.3 Temporal Coherence

There are many situations when one wishes to initialize the local orders in a new list-graph to match those in an existing one – for instance, when moving (Figure 4.13) or editing layers, or when moving the viewpoint around a visibility-edited 3D model. This is nontrivial, as there may not be a one-to-one correspondence between the old regions of overlap and the new ones, and their adjacency may have changed. To overcome this obstacle, I propose an area-weighted voting scheme (illustrated in Figure 4.12). First, regions in the old list-graph send votes for their current stacking order to regions in the new list-graph. These votes are weighted by the area of image-space overlap between the regions. (One can imagine using some sort of pixel-pixel correspondence or motion estimation to warp these regions; however, since I want to handle arbitrary changes, I do not.) Next, these local votes are spread between regions based on consistency. Finally, the new list-graph is ordered by greedily choosing a consistent set of votes.

At each list L with associated area $A(L)$ in the image, I define the reward for a given order $a < b \in L$ as the sum of areas of overlap with old lists L_{old} which have $a < b \in L_{\text{old}}$:

$$R_{\text{local}}(a < b \in L) = \sum_{L_{\text{old}} \text{ s.t. } a<b\in L_{\text{old}}} \text{Area}(A(L) \cap A(L_{\text{old}})) \tag{4.1}$$

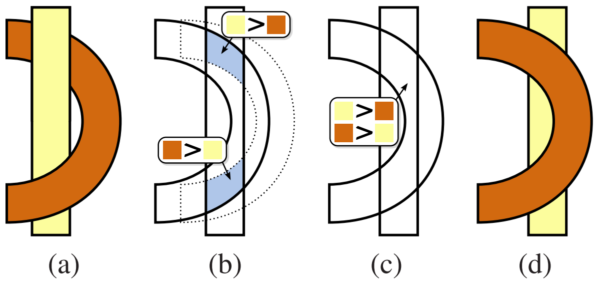


Figure 4.14: *Starting with configuration* **(a)** *the user moves the crescent to the left. The votes,* **(b)**, *are inconsistent in the central region,* **(c)**. *Picking a consistent subset will result in popping (one possibility shown in* **(d)**). *Depending on a user toggle, the system can instead roll back the edit.*

Since choosing a given order $a < b \in L$ will force adjacent lists containing a, b to also adopt that order, I sum the local rewards over each set of adjacent lists:

$$R_{\text{global}}(a < b \in L) = \sum_{L_{\text{adj}} \in C} R_{\text{local}}(a < b \in L_{\text{adj}}) \tag{4.2}$$

Where C is all lists L_{adj} for which, by consistency, $a < b \in L \Rightarrow a < b \in L_{\text{adj}}$; i.e. those L_{adj} such that a path exists from L to L_{adj} with every list in that path containing a, b.

Finally, the system chooses facts (i.e. statements of the form $a < b \in L$) about the ordering in a greedy manner, choosing, at each round, the largest fact that is allowed, given those already chosen:

$$fact_i \equiv \text{argmax}_{\text{allowed } a<b\in L} R_{\text{global}}(a < b \in L) \tag{4.3}$$

(I call an ordering fact "allowed" if it does not contradict a fact that can be derived from those already chosen.)

If any fact with nonzero reward is not allowed, this indicates that the system is unable to perfectly satisfy temporal coherence and some layer will "pop" through another (as in Figure 4.14). Since this may not be desired, I provide the option to halt at this point and roll back the change; this has the effect of stopping layers from being moved through each other and impossible figures from being rotated improperly. Otherwise, the system proceeds as follows.

Once all facts with nonzero rewards are either chosen or not allowed, the system orders each list L based on those facts that can be inferred from the chosen facts. In cases when neither $a < b \in L$ nor $b < a \in L$ can be derived – such as when user actions introduce a new overlap – the program chooses the relative order of a and b to be consistent with a global tie-breaking ordering (if it were to chose locally, it might introduce inconsistencies). This ordering depends on the application. When moving layers, I place the layer being moved at the top of the ordering, so it slides "on top" of anything it is dragged into. When working with depth-peeled 3D models I use the depth ordering, so any new depth complexity winds up stacked properly.

4.4 Extensions

I can extend the notion of local layering beyond simple stacks of static images. In this section, I demonstrate how to adjust layering in animations (viewing each layer as a spatio-temporal volume), and how to use local layering to adjust depth order in 3D models.

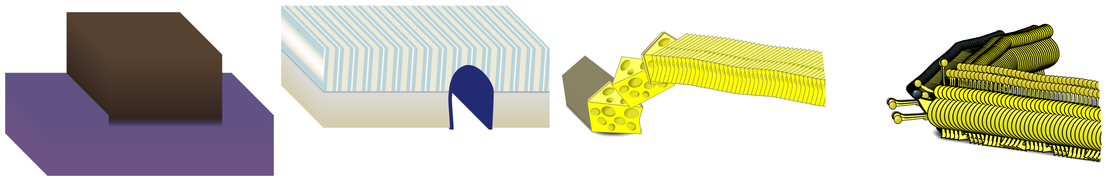


Figure 4.15: *Spatio-temporal volumes corresponding to the four layers used in the animation editing example in Figure 4.16.*

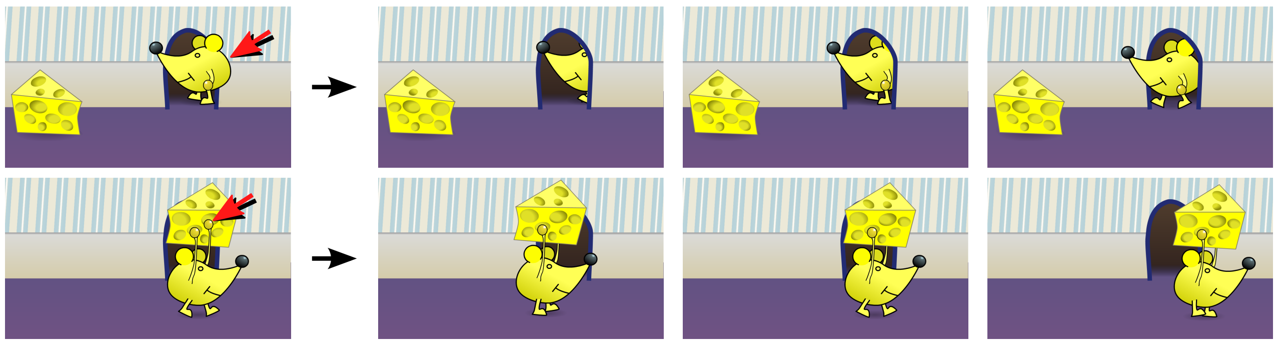


Figure 4.16: *Animation stacking with spatio-temporal overlap regions.* **Top:** *Re-ordering of the mouse and wall at the red arrow in frame 22,* **left***, results in a stacking change in the spatio-temporally local overlap region (***right***, frames 17, 22, and 27).* **Bottom:** *Later in the same animation, placing one of the mouse's hands behind the cheese in frame 94,* **left***, changes an overlap region extending to,* **right***, frames 89, 94, and 99. The mouse passes in front of the wall in these frames because the edit at frame 22 is temporally local.*

4.4.1 Animation

I can adjust the spatio-temporally local stacking of animated layers by extending the notion of adjacency across time. That is, I view the entire animation as a stack of overlapping volumes (e.g., Figure 4.15) and build a list-graph over space-time regions of similar overlap within that volume. Working with space-time volumes means that consistency guarantees layers do not "pop" through each other over time. Temporally local stacking is useful in animations when one element starts behind another then passes in front of it; for instance, a character walking through a door (Figure 4.16).

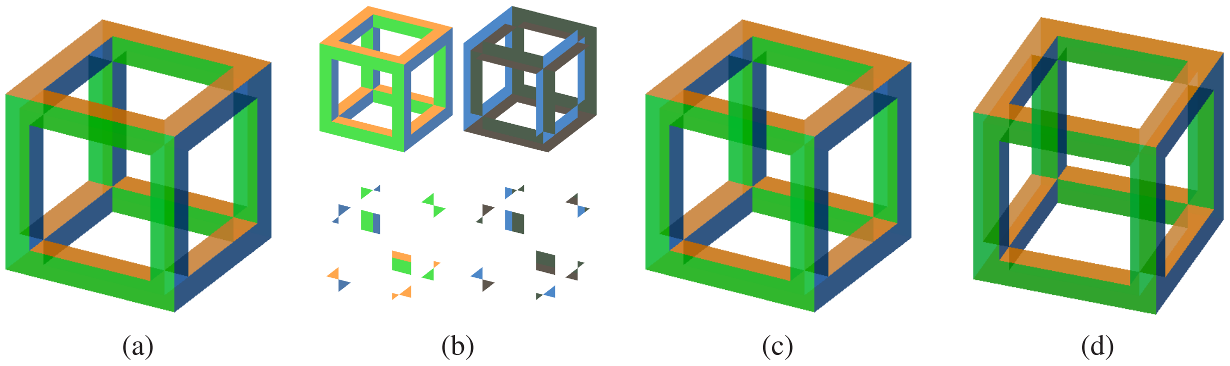


Figure 4.17: *Constructing an impossible cube.* **(a)** *Before manipulation.* **(b)** *Depth-peeled layers.* **(c)** *After local layer rearrangement.* **(d)** *Rotated (stacking preserved using my temporal coherence method).*

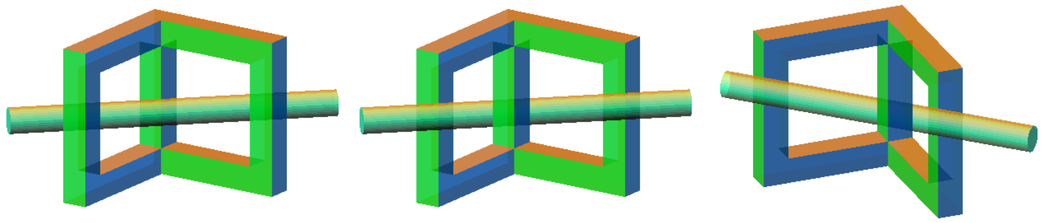


Figure 4.18: *Another impossible figure.* **Left,** *before editing.* **Middle,** *after editing.* **Right,** *another view.*

4.4.2 Impossible Figures

My method of local stacking can also be used to create impossible figures, or correct unwanted interpenetration of 3D objects. In this case, a 3D model is decomposed into layers using GPU-based depth peeling [Everitt 2001], and the user can change the stacking order of these layers using my system. This process is illustrated in Figure 4.17; another example is given in Figure 4.18. Frame-to-frame stacking coherency is maintained using the method of Section 4.3.3.

4.5 Implementation Details

My prototype represents each layer as an image-sized set of pixels with alpha values. The layer is considered to exist wherever it has non-zero alpha. Regions of overlap are calculated by first splatting all the layers into a bitfield image (for example, a pixel overlapped by layers 2, 5, and 7 would have bits 2, 5, and 7 set), then by extracting the connected components of this bitfield. I consider pixels to be adjacent in the up, down, left, and right directions (that is, diagonal adjacency is not considered). When working with animations, pixels immediately before and after a given pixel are considered adjacent as well. A tag image is created that stores the index – in the list-graph – of the stacking order for each pixel. Final compositing proceeds by looking up the stacking order at each pixel (using the tag image) and combining the layers in this order. When working with 3D models, depth-peeling is performed on the GPU and the layers read back into main memory; all other operations are performed on the CPU. While I expect that pushing compositing onto the GPU would accelerate the process significantly, my present CPU version runs fast enough to allow interactive editing of up to 5 megapixel images.

4.6 Discussion

Local layering, as presented herein, is not without its limitations.

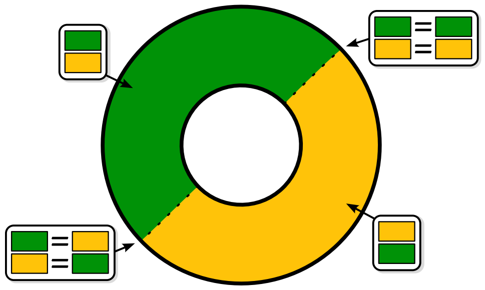


Figure 4.19: *If one augments a list-graph with "gluing instructions" (top right, bottom left) at edges, it is possible to create examples, like the above, which have no consistent orderings.*

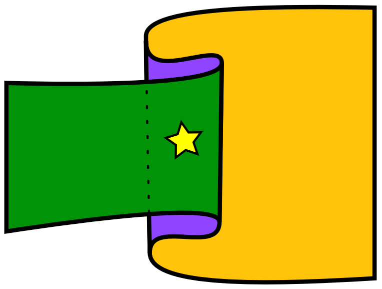


Figure 4.20: *Another problem case for local layering in an environment with self-overlap. If the regions are glued in such a way as to simulate a curled sheet of paper, then there is no consistent stacking that places the purple region on top of the green and yellow regions at the starred location.*

At present, the system cannot handle layers that overlap themselves. To do so I would need to change the list-graph structure to allow a layer to appear multiple times in a given area of overlap. I would also need to add connectivity information to tell the program which instances of layers connect over each edge. However, such list-graphs may not, in general, have any consistent stacking (e.g., Figure 4.19), or, even if they do have a consistent stacking, they may not have a consistent stacking that satisfies a user request (e.g., Figure 4.20), which makes designing algorithms and the associated proofs challenging future work.

While the presented method of creating impossible figures does allow for a neat demo, generating layers with depth peeling is not particularly intuitive. Consider the difficulty a user would encounter in trying to insert a 2D illustration of a curl of smoke rising through the center of the cube in Figure 4.17. Some connected polygons are spread across multiple layers, while other disconnected polygons end up adjacent in the same peel of depth complexity. Splitting the depth-peeled layers at occluding contours and adding connectivity information, as proposed above for self-overlaps, could help address these difficulties. The recent research prototype LayerPaint builds a similar data structure in support of a 3D paint system [Fu et al. 2010].

The presented notion of consistency is founded on layers either being present ($\alpha > 0$) in a overlap or absent ($\alpha = 0$); this works well for layers that represent solid objects, but is not terribly satisfying for objects that are partially transparent everywhere (like fog). To address this, one could envision relaxing my notion of consistency to allow layers to pass through each other smoothly and at a rate proportional to their transparency. This notion is the foundation for the system presented in the next chapter.

While the strength of this approach lies in its locality, this locality can sometimes also be troublesome for users. Consider, for instance, a layer consisting of a cloud of fine particles. To move it in front of another layer requires the user to select each particle individually – a tedious task. To abjure this tedium I could allow multiple regions of overlap to be selected at once, say, by painting a stroke or dragging a box. Local layers dialog operations would then result in invocation of `Flip-Up` or `Flip-Down` at all selected regions. Some care, however, is required if the user selects regions with inconsistent stacking orders – both to convey this inconsistency to the user, and to figure out the "right thing" to do with stacking manipulations.

4.6.1 An Alternate Formulation

While the bulk of this chapter has concerned itself with a recursive formulation of the `Flip-Up` operator, this is not the only possible set-up. An alternate formulation, suggested to me by Gary Miller, involves constructing an object I will call the *consistency graph*. In this section, I briefly sketch this method. It does not seem to offer any practical improvement over the recursive formulation, but is an interesting way to view the problem.

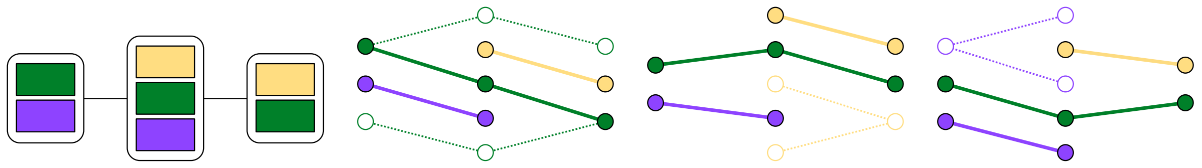


Figure 4.21: *A list-graph (***left***) and the consistency graphs (***right***) for each of the three layers. A consistency graph captures all possible positions of a given layer. In each column possible positions are enumerated. Edges between columns indicate positions that are consistent with each other. Solid dots and lines indicate currently selected positions.*

The consistency graph represents all possible positions of a target layer within the overall stacking. The consistency graph has one vertex for every possible position of the target layer within every list-graph vertex. Those positions in adjacent list-graph vertices that are locally compatible are connected by an edge. A list-graph and corresponding consistency graphs are show in Figure 4.21.

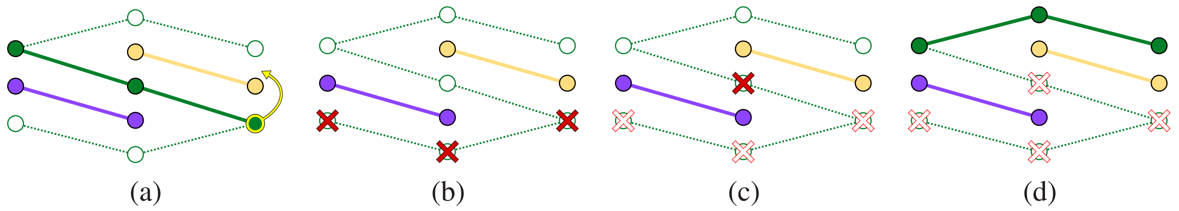


Figure 4.22: **(a)** *The user wants to* `Flip-Up` *the green layer over the tan layer in the right list.* **(b)** `Flip-Up` *should only move the layer up, so all lower possibilities in the consistency graph are eliminated; also, the user wants the green layer to be over the tan layer in the right list, so all positions below the tan layer are eliminated.* **(c)** *Positions that aren't connected to all neighboring lists can't be part of a consistent selection, so are eliminated.* **(d)** *A lowermost position is selected.*

The `Flip-Up` operator can be defined in terms of consistency graph pruning. To move a layer above another layer in a given list, one first constructs the consistency graph for that layer, eliminates possibilities that are not consistent with the desired operation, and – finally – extracts the lowermost remaining position. This process is illustrated in Figure 4.22.

This consistency graph technique, while providing a different view of the `Flip-Up` operation, still seems unable to handle self-overlaps or folds.

4.7 Conclusion

In this chapter, I described a relevant feature in image compositing, local image order, and a local layering system that allows users to edit it interactively. This system relies on operators that preserve the ambiguity of stacking edits – edits propagate only as far as needed to maintain a consistent image, not globally – and in so doing enables the user to easily achieve previously difficult twining and interleaving stacking effects.

Chapter 5

Soft Stacking

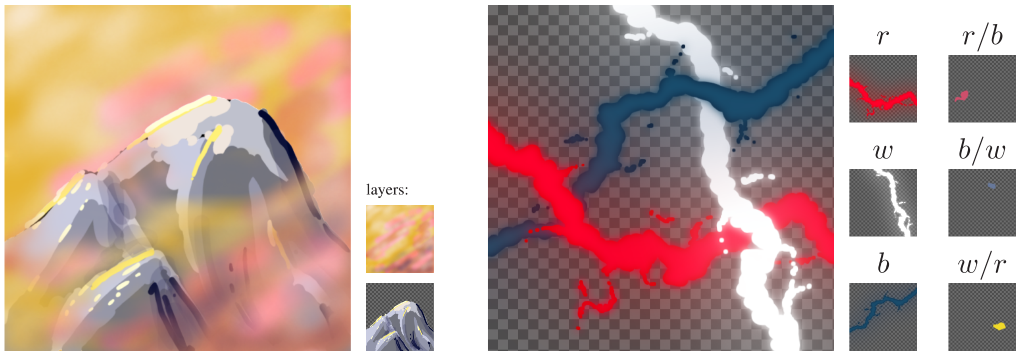


Figure 5.1: *Images stacked with brush-based* **(left)** *and constraint-based* **(right)** *methods for continuous-domain image stacking.*

While Local Layering allows artists to stack paper-like layers according to their intuition, it does not treat volumetric, fog-like layers (e.g., Figure 5.2) in a reasonable way. Unlike paper, fog can pass partially above and partially below other layers.

In this chapter, I introduce the idea of soft stacking, which allows artists to stack fog-like layers intuitively. As per my thesis, soft stacking is built on intuitive local features (local stacking orders), and allows local edits to these features (via a brush that constrains stacking orders). Ambiguity is preserved through locality – the stacking brush only alters stackings exactly where it is being applied.

I also formulate a continuous optimization version of this stacking problem to allow the computer to propagate stacking edits across an entire image in a smooth and reasonable way. This continuous optimization version does not exactly fit with my thesis because it is not real-time; however, I include it for completeness. Nevertheless, the optimization approach does preserve the ambiguity of the artist's intent by introducing as few layer-layer collisions as possible. This limits the stacking drama introduced to that specifically indicated by the artist's constraints. It is telling that the results produced with this optimization are generally less attractive than those produced by the (interactive) order-painting system.

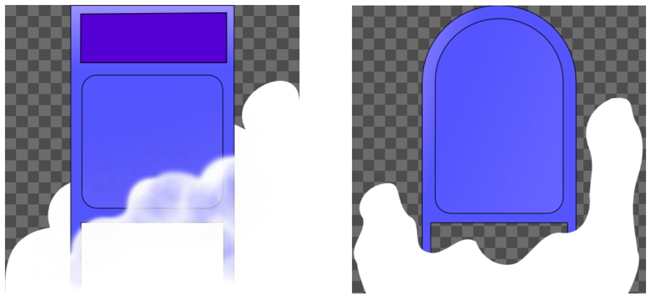


Figure 5.2: Left: *Fog twines about a postbox.* **Right:** *Possible side view; notice some fog is in front of the postbox, and some behind.*

5.1 Contribution

Under a standard probabilistic interpretation of image opacity, any depth arrangement of volumetric layers can be represented as a convex combination of non-interleaved arrangements.

At the core of this work are two editing methods for creating these mixed stackings: a stacking brush which allows artists to paint spatially varying stacking orders; and an optimization technique which interpolates sparse stacking constraints across an image in a way that minimizes collisions between layers.

5.2 History

When creating two-dimensional compositions on a computer, artists often assemble final images from stacks of layers. Layers allow artists to separate semantically different portions of an input scene, adjust gross composition, and achieve occlusion effects. The story of on-computer stacking (and its relation to film compositing) is covered in some detail in the previous chapter.

Of course, in certain scenarios, it is infeasible or artistically undesirable that layers should appear in the same order everywhere in an image. Methods have been proposed for 3D scenes [Duff 1985; Snyder and Lengyel 1998] and 2D images [Wiley 2006] (and the previous chapter) that allow local ordering of layers. Indeed, such methods have even been adopted into commercial software [Mediachance 2001–2009; Asente et al. 2007]. However, these methods still restrict users to selecting exactly one stacking order at every image location, a representation insufficient to model the behavior of layers representing volumetric media.

Compositing volumetric media into scenes is difficult because volumes can appear both in front of and behind other objects simultaneously, and the portion of the volume that is in front of another object can vary spatially. To date, the only methods devised to handle this scenario have been ad hoc or required access to volume data. Ad-hoc methods generally involve duplicating the volumetric layer and carefully manipulating alpha channels. Full-volume methods are quite data intensive, and restrict artists to using computer-generated volumes (general volume capture isn't yet a practical technology).

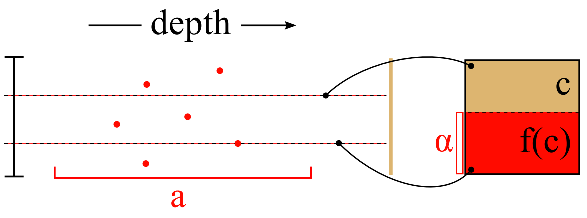


Figure 5.3: *The probabilistic interpretation of image opacity.* **Left:** *When rays are cast through layer a, their probability of interaction is α.* **Right:** *The final color is a combination of the background color, c, and the result of the layer interacting with the background color, $f(c)$.*

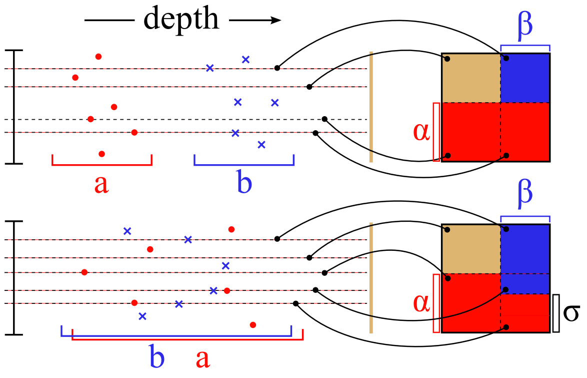


Figure 5.4: *The probabilistic interpretation of image opacity, with two layers. Final color at a pixel can be determined as a combination of five possible sample types – both in standard stackings* **(top)** *and when the layers overlap or interleave in depth values* **(bottom)**.

5.3 Model

I motivate my soft stacking approach by appealing to a standard, probabilistic notion of image opacity, α:

$$\alpha(x) \equiv \mathbb{P}(\text{ray at } x \text{ intersects layer}) \tag{5.1}$$

This model is convenient because it is general; given an arbitrary transfer function, $f(\cdot)$, I can write down the expected color at a pixel by summing the contribution from rays that strike the layer and those that pass by and strike the background color c_{in}:

$$\mathbb{E}[c_{\text{out}}] \equiv \alpha f(c_{\text{in}}) + (1-\alpha)c_{\text{in}} \tag{5.2}$$

A visual interpretation is given in Figure 5.3.

A simple case. When two layers (with opacities α_1, α_2 and transfer functions f_1, f_2) are present, I can again write down the expected color as a sum over possible samples. However, I need to introduce two new parameters, $\sigma_{1/2}$ and $\sigma_{2/1}$, that tell us the proportion of rays that hit layer 1 first among those rays that hit both layers 1 and 2:

$$\mathbb{E}[c_{\text{out}}] \equiv \begin{array}{ll} (1-\alpha_1)(1-\alpha_2)c_{\text{in}} & \text{miss both} \\ +\alpha_1(1-\alpha_2)f_1(c_{\text{in}}) & \text{hit layer 1} \\ +(1-\alpha_1)\alpha_2 f_2(c_{\text{in}}) & \text{hit layer 2} \\ +\sigma_{1/2}\alpha_1\alpha_2 f_1(f_2(c_{\text{in}})) & \text{hit layer 1 before 2} \\ +\sigma_{2/1}\alpha_1\alpha_2 f_2(f_1(c_{\text{in}})) & \text{hit layer 2 before 1} \end{array} \tag{5.3}$$

A visual interpretation is given in Figure 5.4. Notice that, whatever the depth interleaving, I can describe it with $\sigma_{1/2} \geq 0$ and $\sigma_{2/1} \geq 0$ (it isn't possible for fewer than zero rays to hit something), and $\sigma_{1/2}+\sigma_{2/1} = 1$ (all rays that hit both must be accounted for). That is, I can account for any depth interleaving with a convex combination of stacking coefficients σ.

Some notation. In general, I will discuss a set of L layers labeled $1, \ldots, L$ with transfer functions $f_1, \ldots, f_L$. Let $i/j/k$ represent the stacking i over j over k, the shorthand $\bullet$ or $\star$ represent an arbitrary such stacking, and $\text{perm}(L)$ be the set of all such stackings.

The general statement. The expected output color of an arbitrary depth-mixing of volumetric layers is a convex combination of the expected colors of the possible non-mixed orderings:

$$\mathbb{E}[c_{\text{out}}] = \sum_{\bullet \in \text{perm}(L)} \sigma_\bullet \mathbb{E}[c_\bullet] \qquad \text{with} \quad \begin{array}{l} \forall \bullet : \sigma_\bullet \geq 0 \\ \sum \sigma_\bullet = 1 \end{array} \tag{5.4}$$

Where $c_\bullet$ is a random variable that takes the color of each possible combination of layer interactions with the probability of that combination. Specifically, I define

$$c_{i/\bullet} \equiv \begin{cases} f_i\left(c_\bullet\right) & \text{with probability } \alpha_i \\ c_\bullet & \text{otherwise} \end{cases} \tag{5.5}$$

$$c_i \equiv \begin{cases} f_i\left(c_{\text{in}}\right) & \text{with probability } \alpha_i \\ c_{\text{in}} & \text{otherwise} \end{cases} \tag{5.6}$$

Thus, to allow a user to edit a stack of layers as if they were volumes of fog, I must provide them with tools for selecting convex combinations of the stacking coefficients, $\{\sigma_\star\}$. In the remainder of this chapter I will discuss two such methods: brush-based editing lets the artist paint stacking orders, while optimization-based editing interpolates sparse order constraints to arrive at stacking orders for every pixel.

5.4 Brush-Based Editing

In my brush-based stacking editor, I allow the artist to edit the convex combination of stacking coefficients at each pixel directly. To do so, she first selects an ordering constraint using an ordering widget (Figure 5.5), then paints on the image. Wherever she paints, the stacking coefficients are modified to respect the given constraint, with the strength of the modification being proportional to the opacity of the brush stroke.

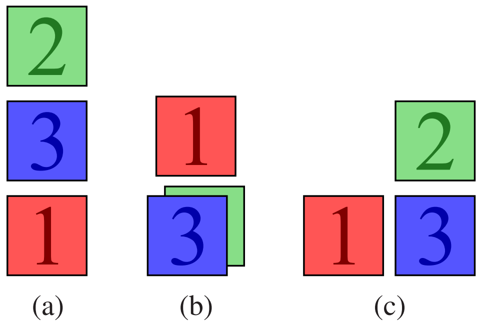


Figure 5.5: *Three configurations of my ordering widget. Each column instructs my program on the relative ordering of the layers in that column.* **(a)** *"Place layer 2 over layer 3 over layer 1."* **(b)** *"Place layer 1 on top; don't change the order of layers 2 and 3."* **(c)** *"Place layer 2 over layer 3. Do not change the position of layer 1."*

$$\underset{(a)}{\begin{bmatrix} 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 1&1&1&1&1&1\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0 \end{bmatrix}} \quad \underset{(b)}{\begin{bmatrix} 1&0&1&1&0&0\\ 0&1&0&0&1&1\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0 \end{bmatrix}} \quad \underset{(c)}{\begin{bmatrix} 1&1&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&1&0&1&0\\ 0&0&0&1&0&1\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0 \end{bmatrix}} \quad \underset{(d)}{S \equiv \begin{bmatrix} \sigma_{1/2/3}\\ \sigma_{1/3/2}\\ \sigma_{2/1/3}\\ \sigma_{2/3/1}\\ \sigma_{3/1/2}\\ \sigma_{3/2/1} \end{bmatrix}}$$

Figure 5.6: **(a-c)** *Matrix C such that CS is consistent with the constraints specified in Figure 5.5.* **(d)** *The ordering of S used when computing C.*

Concretely, this means that column vector $S_x = [\sigma_{1/\bullet/L} \ldots]^T$ of stacking coefficients for pixel x is modified based on the stroke opacity $\beta(x)$ at that pixel as follows:

$$S_x \leftarrow (1 - \beta(x))S_x + \beta(x)CS_x \tag{5.7}$$

The C matrix encodes the stackings-to-stackings mapping specified by the user with the ordering widget. Since C maps each input stacking coefficient to one output coefficient and does not scale, it is easy to see that (5.7) maintains the convexity of S. I give in Figure 5.6 the C matrices corresponding to the ordering constraints in Figure 5.5.

5.4.1 Implementation

I wrote my prototype brush-based stacking editor in C++ using OpenGL for layer compositing. It is able to handle painting stacking orders for moderate numbers (3-4) of relatively small (512x512) layers in real-time on an everyday system (Pentium4 3.2GHz, GeForce 8400GS), and 5 layers on a workstation (Core2 Quad Q6600, GeForce 9800GX2). The limiting factors for system performance are video memory and fill rate. This is because the system uses one 8-bit/pixel greyscale texture for each of the $L!$ stacking coefficients, and

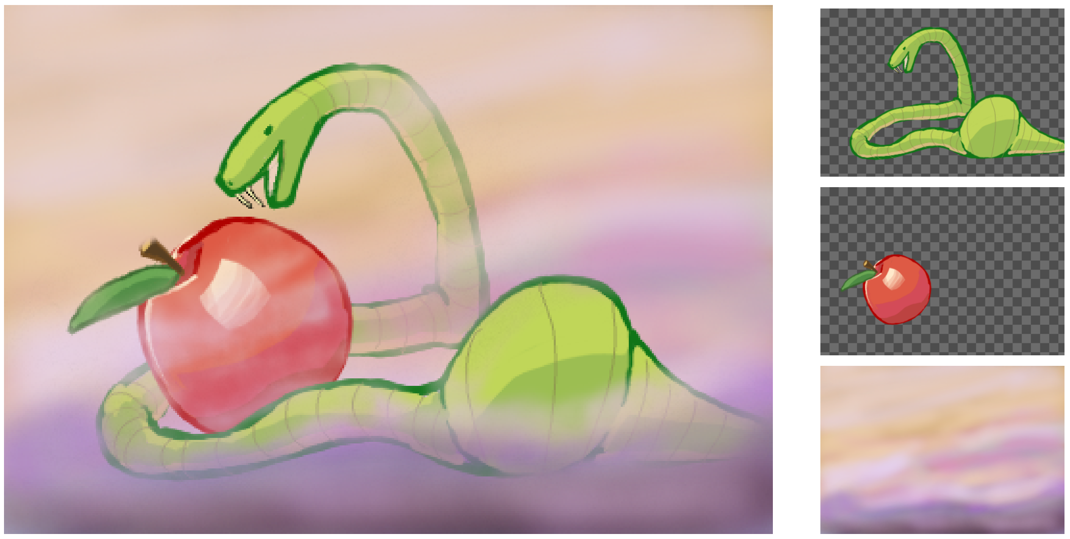


Figure 5.7: *A brush-based stacking result with three layers. Notice the local ordering of the snake and apple as well as the snake, apple, and mist.*

needs to compute, weight, and accumulate $L!$ stacking orders to produce a final image. I believe that both of these factors could be mitigated to a certain extent by tiling the input layers and only computing on those tiles that are currently being painted. Such a tactic would allow much larger images to be used; however, the factorial complexity in the number of layers means that any system will likely be limited in that regard.

5.4.2 Results

I find that my brush-based stacking system is useful for adding fog- or mist-like effects to a scene (Figure 5.1, left; Figure 5.2, left). It can also be used in situations where previous discrete layering approaches would be useful (the interleaving of snake and apple in Figure 5.7).

5.5 Optimization-Based Editing

Though painting layering coefficients is direct and intuitive, sometimes an artist may wish to provide a more sparse set of inputs, e.g., “Layer 1 should be on top on the left of the image and layer 2 should be on top on the right.” In this case, the system should be able to determine an optimal selection of stacking coefficients subject to these user constraints. Intuitively, an optimal set of stacking coefficients should change smoothly, and should allow less-opaque layers to pass through each other more readily.

One can do this by selecting stacking coefficients that minimize a sum of penalties (Figure 5.8) between

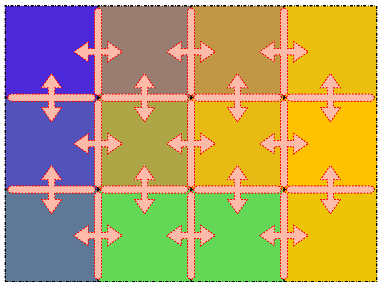


Figure 5.8: *My system picks stacking coefficients at each pixel to minimize the sum of the penalties between all pairs of adjacent pixels.*

adjacent pixels x, x':

$$S \leftarrow \operatorname*{argmax}_S \sum_{x,x'} P_\alpha \left(S_x - S_{x'}\right) \tag{5.8}$$

Where S contains all the stacking coefficients in the image; S_x is the column vector containing the stacking coefficients for pixel x; and P_α is a penalty function dependent on layer opacity that I will construct presently.

Notice that if I were to set $P_\alpha(\Delta) = ||\Delta||^2$, (5.8) would reduce to Poisson interpolation. This does satisfy our intuition about smoothness, but it does nothing to differentiate between layers of different opacity. My model of opacity suggests that whenever layers i and j pass through each other, there is a $\alpha_i\alpha_j$ probability of a collision; this provides a natural starting point for a penalty function:

$$P_\alpha(\Delta) \equiv \sum \alpha_i\alpha_j(\text{amount of } i \text{ through } j \text{ required by } \Delta)^2 \tag{5.9}$$

In order to determine how much of layers i and j must pass through each other between two pixels, I construct and solve for an optimal flow along a graph of stackings (Figure 5.9). Each vertex in the graph is a possible stacking order, and each edge corresponds to a transposition of two adjacent layers:

$$V \equiv \text{perm}(L) \tag{5.10}$$

$$E \equiv \{(\bullet/i/j/\star, \bullet/j/i/\star) \quad | \quad i < j\} \tag{5.11}$$

Now, given stacking coefficients σ and σ' at adjacent pixels, I can determine a set of flows along edges, δ, which are consistent with the change in stacking order between them:

$$\forall \star \in V : \sigma_\star - \sigma'_\star = \sum_{(\bullet,\star)\in E} \delta_{\bullet,\star} - \sum_{(\star,\bullet)\in E} \delta_{\star,\bullet} \tag{5.12}$$

Since, when $L > 2$, there is more than one selection of δs which satisfy the difference in stackings, I select a flow that results in the minimum penalty:

$$P_\alpha(\sigma - \sigma') \equiv \min_\delta \sum_{(\bullet/i/j/\star,\bullet/j/i/\star)\in E} \alpha_i\alpha_j \left(\delta_{\bullet/i/j/\star,\bullet/j/i/\star}\right)^2 \tag{5.13}$$
$$\text{subject to (5.12)}$$

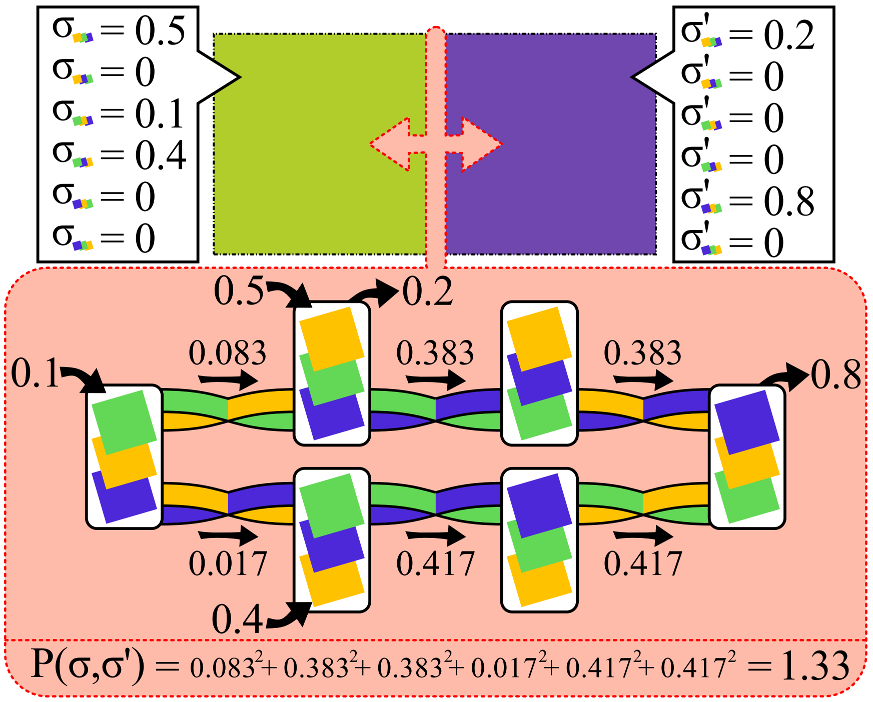


Figure 5.9: *I penalize the change in stacking order coefficients, σ, between adjacent pixels by charging for the minimum-weight flow through a graph of stackings. (Each edge is weighted by the product of the opacities of the swapped layers; in this example all pixels are fully opaque, so no weights are shown.)*

Finally, since $P_\alpha(\sigma - \sigma')$ is the result of a least-squares solve with linear equality constraints, I can re-write the objective function (5.8) as a constrained quadratic minimization:

$$S \leftarrow \underset{S}{\text{argmax}} \sum_{x,x'} \left|\left| W_\alpha \left(S_x - S_{x'} \right) \right|\right|^2 \quad (5.14)$$

$$\text{with } S_x \text{ convex}$$

Where W_α is the weighting matrix derived from P_α, and the opacities in α are sampled at $\frac{1}{2}(x + x')$.

5.5.1 Implementation

My objective function (5.8) is a quadratic optimization with inequality constraints (due to the convexity requirement on the stacking coefficients, σ). Unfortunately, the scale and sparsity of the system prevented me from using an off-the-shelf quadratic program solver.

Instead, I use a form of gradient descent; alternating steps of successive over-relaxation (which bring the solution closer to the global, unconstrained minimum) and constraint enforcement. Steps are taken until a fixed limit is reached or the maxiumum change drops below a specified threshold. For examples in this chapter I use a limit of 1000 iterations and a threshold of $\frac{1}{2560}$.

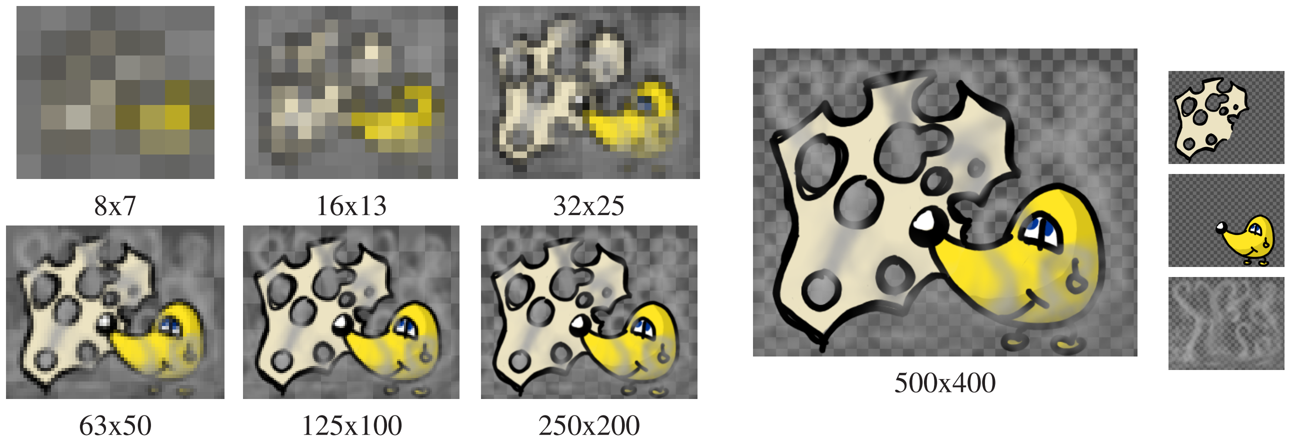


Figure 5.10: Left: *My solver iteratively optimizes at progressively higher resolutions.* **Right:** *The final smoked-cheese image and source layers.*

To aid in convergence, I start by scaling down the layers and constraints and iterating on this very coarse version of the problem, then project this solution to finer and finer levels – refining after each projection (Figure 5.10). This is similar to a multigrid approach[1], except that I never perform any further cycles or attempt to restrict residual error (the inequality constraints would make this difficult to do).

My solver took 214 seconds on a workstation-class computer (Core2 Quad Q6600) to produce the 512x512, 3-layer result shown in Figure 5.1, with about a quarter of that time being taken on weighting matrix construction and three quarters on iteration. A more sophisticated solution technique (e.g. algebraic multigrid) might result in better convergence and thus reduce solution time.

5.5.2 Results

My optimization-based approach provides reasonable stacking order interpolation in a number of situations. It is useful when interleaving layers with volumetric glows (Figure 5.1, right), and smoke-like layers (Figures 5.10, 5.11). It works on drawn as well as photographic (e.g. Figure 5.12) source material. Though the optimizer is not interactive, oftentimes the low-resolution intermediate results are enough to let one know that something is not stacking as intended.

5.6 Discussion

In this chapter, I demonstrated two approaches for the creation of soft stackings – arrangements of layers as if they were made of volumetric media. My methods are motivated by a standard probabilistic notion of opacity and work for any layer transfer functions. The brush-based stacking method has the advantage of

[1]In fact, some people call this cascadic multigrid; I call that spackling over a hack with linguistic loop-de-loops.

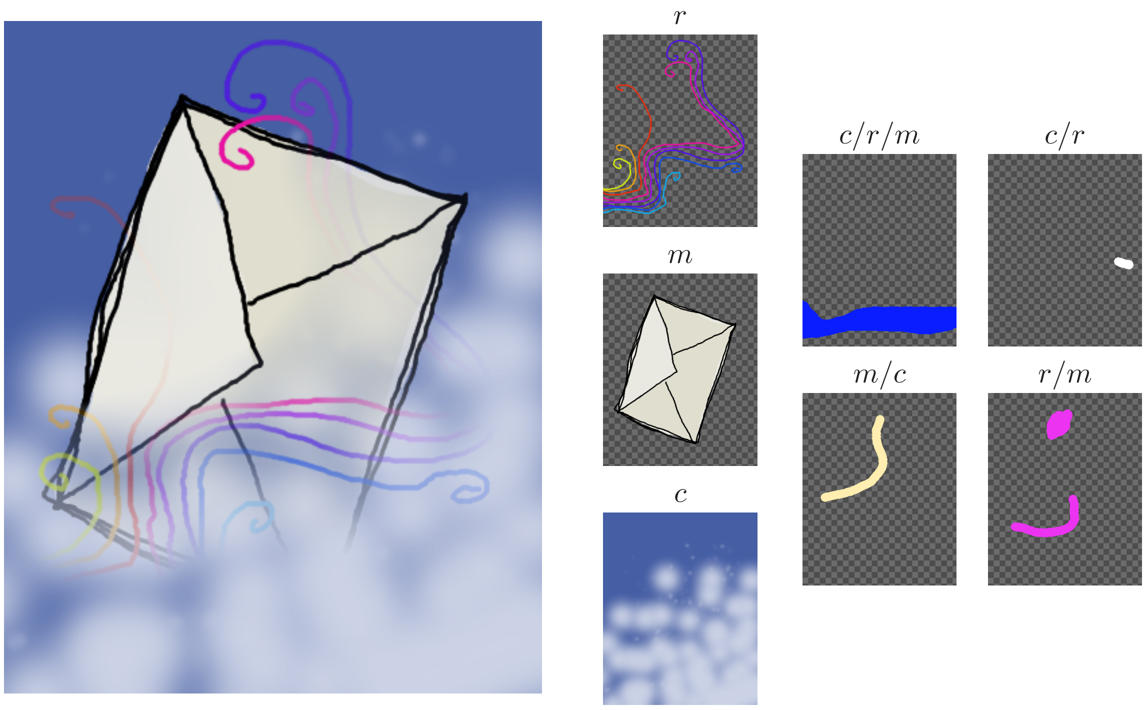


Figure 5.11: *A result generated with my optimization-based approach. The color used in constraint images* **(right)** *is ignored.*

allowing direct control and providing rapid feedback, while the optimization-based stacking approach can provide a globally smooth result given sparse constraints.

While both my proposed methods require $L!$ stacking coefficients, modern computers have enough power and storage to handle stacking tasks of reasonable size. Indeed, it seems unlikely that detailed interleavings of even five volumetric layers have substantial artistic use.

Additionally, if many layers must be stacked, a subset of the brush-based stacking method (only allowing explicitly selected orderings) can be implemented by providing what is essentially a clone brush from a "virtual ordering" of layers. This approach relies on the fact that a linear transformation (weighted color sum) of a convex space (the stacking coefficients) is itself convex.

I have tried formulating the penalty function for the optimization-based stacking based on summed expected collisions (that is, replacing the $(\cdot)^2$ with $|\cdot|$ in (5.13)). The resulting linear program solution was not visually pleasing, as it tended to concentrate the error at a few pixels.

In the future, I would like to investigate methods for reducing the storage cost, perhaps by storing coefficients on a coarse mesh instead of per pixel, or by using a reduced basis for stacking coefficients. My optimization system should also trivially extend to animations, though the required memory might be quite high.

Overall, my notion of soft stacking extends the current state of the art in layer stacking by formulating the

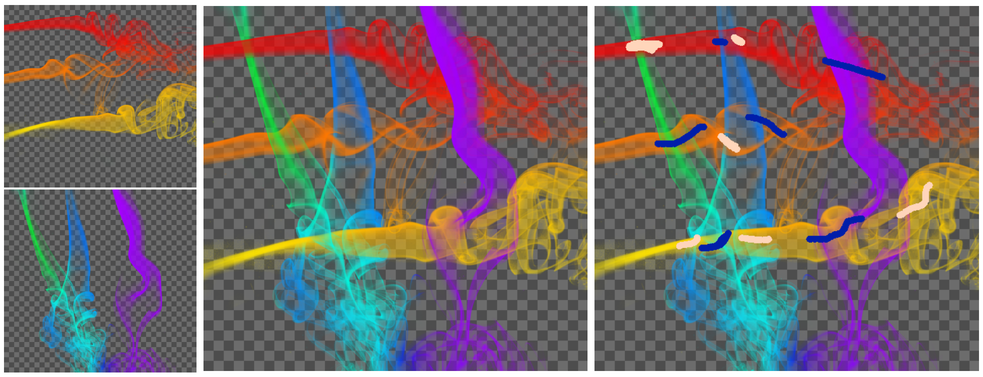


Figure 5.12: *Optimized stacking result with smoke. In the constraint image* **(right)**, *light pink is warmer-colored layers on top and dark blue is cooler-colored layers on top. (Source photograph from aubergene via flickr.)*

problem in the continuous domain. This provides artists with the ability to manipulate layers as if they were volumes of fog, without having to explicitly construct volume data.

Soft stacking is yet another example of a more powerful image editing tool built on intuitive local features ("in front" relationships) and real-time feedback.

Chapter 6

3D-like Texturing for 2D Animation

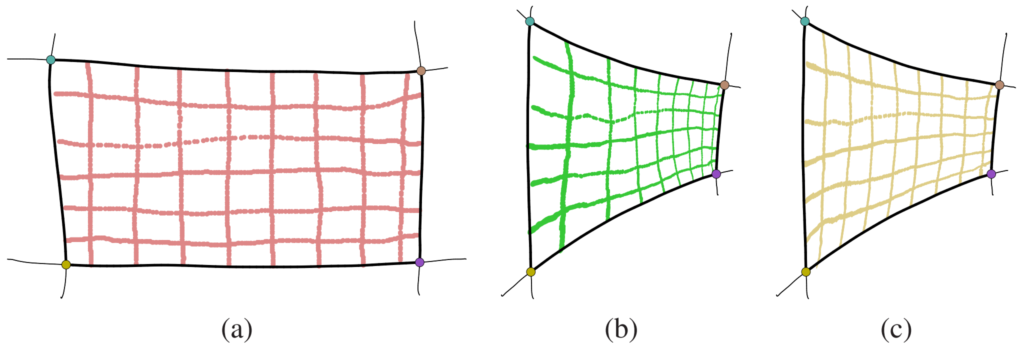


Figure 6.1: *My system allows artists to add shape cues to help in transferring textures between animation frames. Here, the source paint,* **(a)**, *is transferred to a slanted wall,* **(b)**. *Without shape cues, perspective effects go missing,* **(c)**.

One challenging aspect of 3D shape depiction in 2D animation is applying surface textures to objects. This is because a surface texture must move in a way that is plausible given the (3D-like) motion of an animated surface. In this work, I present a system that addresses this problem and – in keeping with the theme of this document – does so by relying on local, intuitive, artist-specified surface shape cues. Further, my system preserves much of the ambiguity of the source animation by avoiding the creation of 3D geometry.

This system is, however, imperfect; the results are not as stunning as one might hope and the user experience is unpolished. However, I think it is a reasonable experiment and sheds light on an interesting future direction for graphics research.

6.1 Contribution

The main contribution of this section is a 3D-model-less method for defining surface correspondences in 2D animations, based on artists' local surface-warp specifications.

6.2 Details

Perceptually, surface texture is an important cue. It can help a viewer estimate surface shape and properties (e.g. where the surface stretches or bends).

Naturally, then, 2D animators may wish to include surface texture in their works. The trouble with surface texture is that it is a dense phenomenon, so specifying it by hand is tedious, and specification errors are readily apparent. (Contrast this to lighting, e.g. Figure 6.2, which is often approximated by dividing an object into "light" and "dark" regions, and for which errors – at least systematic ones – are not particularly noticeable.)

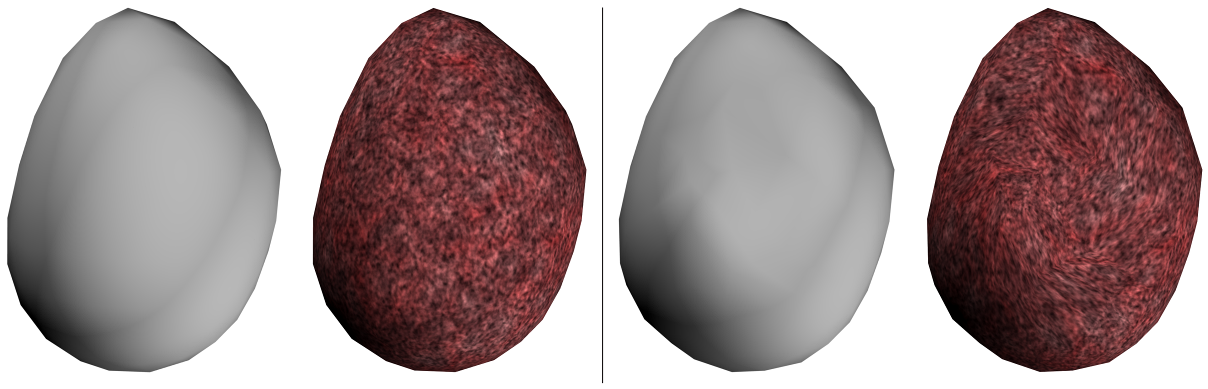


Figure 6.2: *Estimating geometry for texturing is difficult because surface texture reveals more about geometry than lighting. If an artist was thinking of the geometry on the left and an algorithm produced the geometry on the right, it would likely be sufficient for lighting, but highly unsatisfactory for texturing (notice the swirls). (Note: both geometries in this example constructed by hand.)*

The key observation of this work is that by providing artists with good tools for producing a sparse description of local shape my system can create frame-to-frame correspondences that make surface texturing easy – without ever creating a 3D model. Because the system described herein does not create a 3D model, it doesn't rely on any particular projection system or knowledge about global object deformations. Instead, both of these properties are combined into an estimate of "surface warp," which the artist is able to specify with local gizmos. Moreover, my system doesn't demand that the artist specifies surface warp values everywhere or fully accurately – all that is required is that the patch of the object on which a texture is to appear is sufficiently described by the artist to achieve the deformation he desires. In this way, any shape information not relevant to the artist's goals is left ambiguous.

Figure 6.3: *For the purposes of this work, I define a texturing of an animation as a dense correspondence between screen-space points in the animation. Points may not have corresponding points in all frames.*

6.3 History

The research prototype Textureshop [Fang and Hart 2004] attempts to provide 3D-like texturing for images by running texture synthesis over a local 3D reconstruction from image normals. As it is a texture synthesis approach, direct artistic control is limited. It is also not designed for animations, though one could imagine adding temporal coherence constraints to the synthesis algorithm.

If a 3D model of a scene exists, texturing can be calculated by warping that model to match 2D contours [Corrêa et al. 1998]. Unfortunately, such an approach requires that an animation be created twice – once as a 2D animation and once as near-matching 3D animation. A simpler variant of this approach appears in Photoshop [Adobe 2008], which allows artists to paint on a 3D model (though no warping of the model is allowed).

Where my work adds a smidgen of 3D-like information to 2D drawings, the opposite (adding some 2D-like effects to 3D objects) has also been explored in graphics; e.g. by creating geometry that changes depending on the viewing direction [Rademacher 1999], or by calling on artistic inspiration in creating and rendering 3D models [Markosian 2000].

3D or 2.5D[1] models have been used for other cartoon manipulations as well. A simple billboards-at-points model is sufficient to produce rotatable cartoon figures [Rivers et al. 2010b], or perform tweening [Fiore et al. 2001]. An approximate 3D model – akin to the tubes-and-spheres construction lines used in traditional drawing – is useful when performing computer-assisted tweening [Fiore and Reeth 2002].

My work makes use of sparse user input to specify local surface shape. Previous systems have used similar input to derive global depth maps. Sparse depth constraints can be interpolated to provide depth information for posing or rendering stereo views of cartoon figures [Sýkora et al. 2010]. Sparse depth specification tools can also be employed for single view modeling [Zhang et al. 2001]. The work I will presently describe is different in that it moves directly from a (possibly inconsistent) local surface model to a frame-to-frame correspondence without using any depth information.

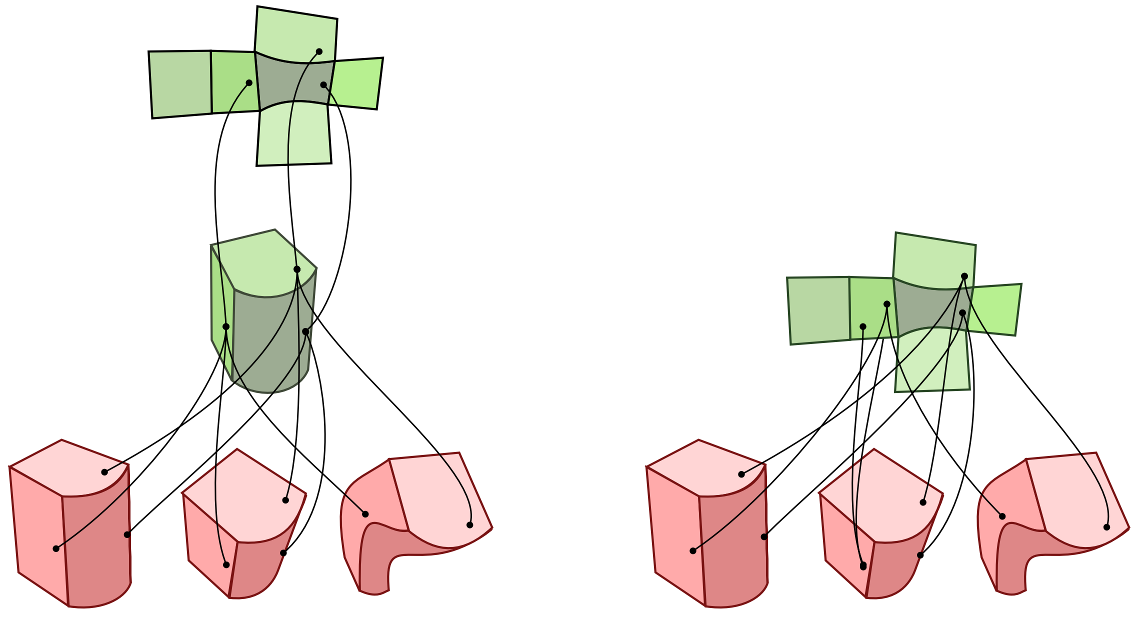


Figure 6.4: Left: *Traditional 3D texturing defines a dense correspondence between 2D frames (bottom) and a reference model (middle) that is in correspondence with a 2D texture atlas (top).* **Right:** *My texturing approach places the 2D frames (bottom) directly in correspondence with a texture atlas (top).*

6.4 Defining Texturing

Before I can talk about how I will go about texturing 2D animations, I need to define what I mean by texturing:

Definition 6.4.1 (Texturing)**.** A *texturing* of a series of frames is a correspondence between points in those frames (with the possibility that some points do not have analogous points in other frames due to, e.g., occlusion).

This definition is illustrated in Figure 6.3.

Conventional 3D texturing implements this notion by providing a correspondence between a known 3D model and each frame (Figure 6.4, left). However, it is important to notice that my definition does not require or imply that such a model must exist.

Of course, simply coming up with some texturing (a trivial one: don't provide any correspondences between frames) isn't going to be enough. Indeed, my algorithm aims to produce:

Definition 6.4.2 (Correct Texturing)**.** A *correct texturing* is one that matches a user's desired point correspondences for all points relevant to the current task.

[1]2.5D $\equiv$ 2D with basic overlap and/or relative depth information

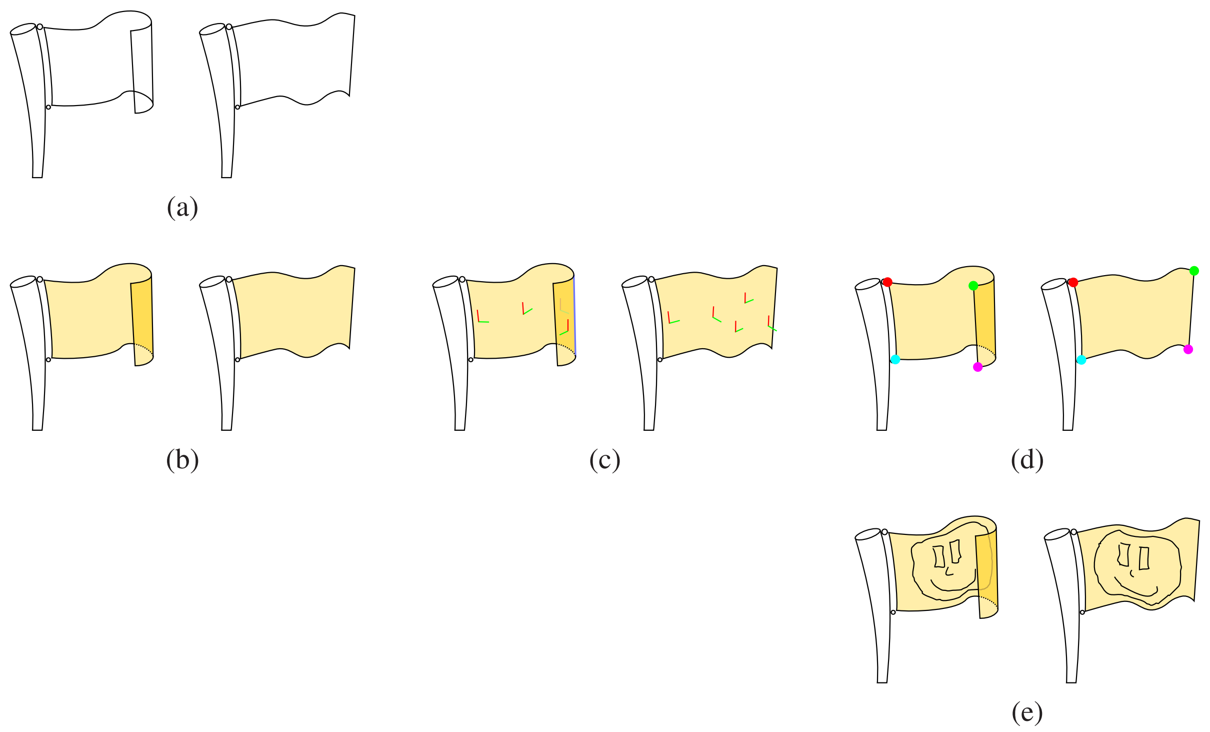


Figure 6.5: *A cartoon of the system's texturing workflow. Artists begin with* **(a)** *a series of animation frames; specify* **(b)** *a 2.5D paneling,* **(c)** *shape constraints, and* **(d)** *a sparse correspondence (colored circles); and the system uses this information to* **(e)** *transfer surface detail.*

Notice, that while a conventional 3D texturing approach certainly satisfies the first definition, the second – more user-centric – constraint may not be met.

6.5 Workflow

My animation texturing system works within a rudimentary planar-map illustration interface. I begin by describing the texturing workflow from an animator's point of view, as shown in Figure 6.5.

The animator begins by drawing a sequence of frames containing a region he wishes to add texture to. He then uses the gluing tool to create a 2.5D paneling of this region by attaching adjacent planar map faces along common edges. Next, he specifies shape information by using two local constraints – warp constraints (which specify the projection of u and v vectors on the surface) and contour constraints (which are automatically created along occluding contours, and require that the projections of the surface tangents are tangent to to the contour[2]). Finally, he specifies a sparse correspondence between shape boundaries in each frame by selecting a few feature points along each boundary.

[2]This is because the normal to the surface is perpendicular to the viewing direction at a contour, so the tangent plane of the surface projects to a tangent line along the contour.

The system then creates a mapping that respects these constraints, and the animator's surface detail strokes automatically transfer from frame to frame.

6.5.1 Shape Constraints

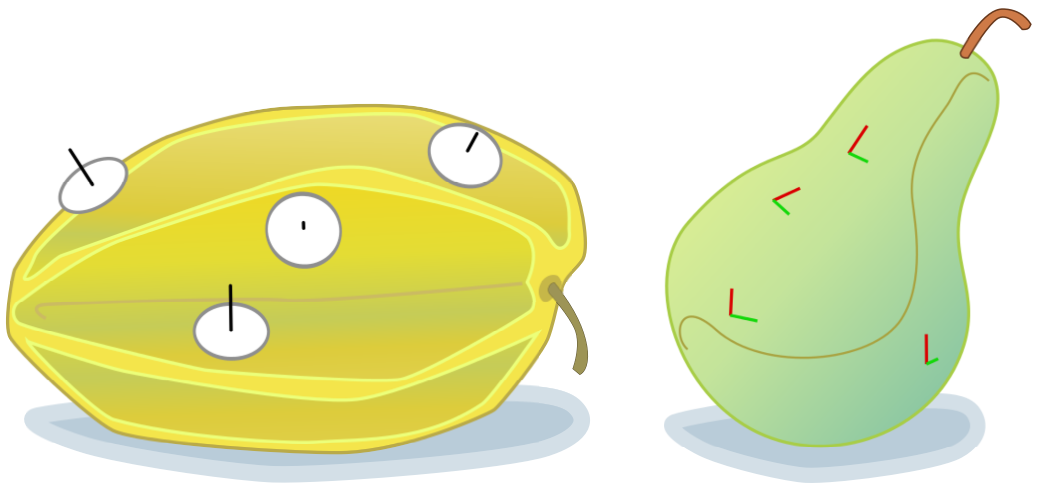


Figure 6.6: **Starfruit**, *the standard normal-representing circle-and-stick.* **Pear**, *my surface-warp-representing axis pairs.*

While drawing and gluing are straightforward operations, shape constraint specification – as a crucial part of this system – requires a more full explanation. One standard method of local shape specification is normal specification, in which the user positions a thumbtack-like gauge to visually align with the surface (Figure 6.6, starfruit). In a perceptual study, this method was reported by the subjects to be the "most natural" of several tried (including point comparison and direct depth specification) [Koenderink et al. 2001].

However, while normal specification is sufficient for shape representation in pictures with near-orthographic cameras, extreme perspective distortions can make the figure hard to properly align. Worse still, such distortions are very hard to estimate from just normals. For these reasons, I chose to use a pair of axes as a surface constraint (Figure 6.6, pear). These axes are adjusted by the user until they lie in the plane of the object, aligned in some user-imagined consistent coordinate directions.

Contours (edges where a back-facing and front-facing panel are glued together) are also implicitly shape constraints, and I will treat the effect of these later, in the shape determination section.

6.6 Method

In order to support this workflow, my system needs to perform three important tasks: first, it must interpret the artist's shape constraints to understand surface shape; second, it must use this understanding of surface shape to (locally) hypothesize a source shape; finally, it must create a mapping between these source shapes.

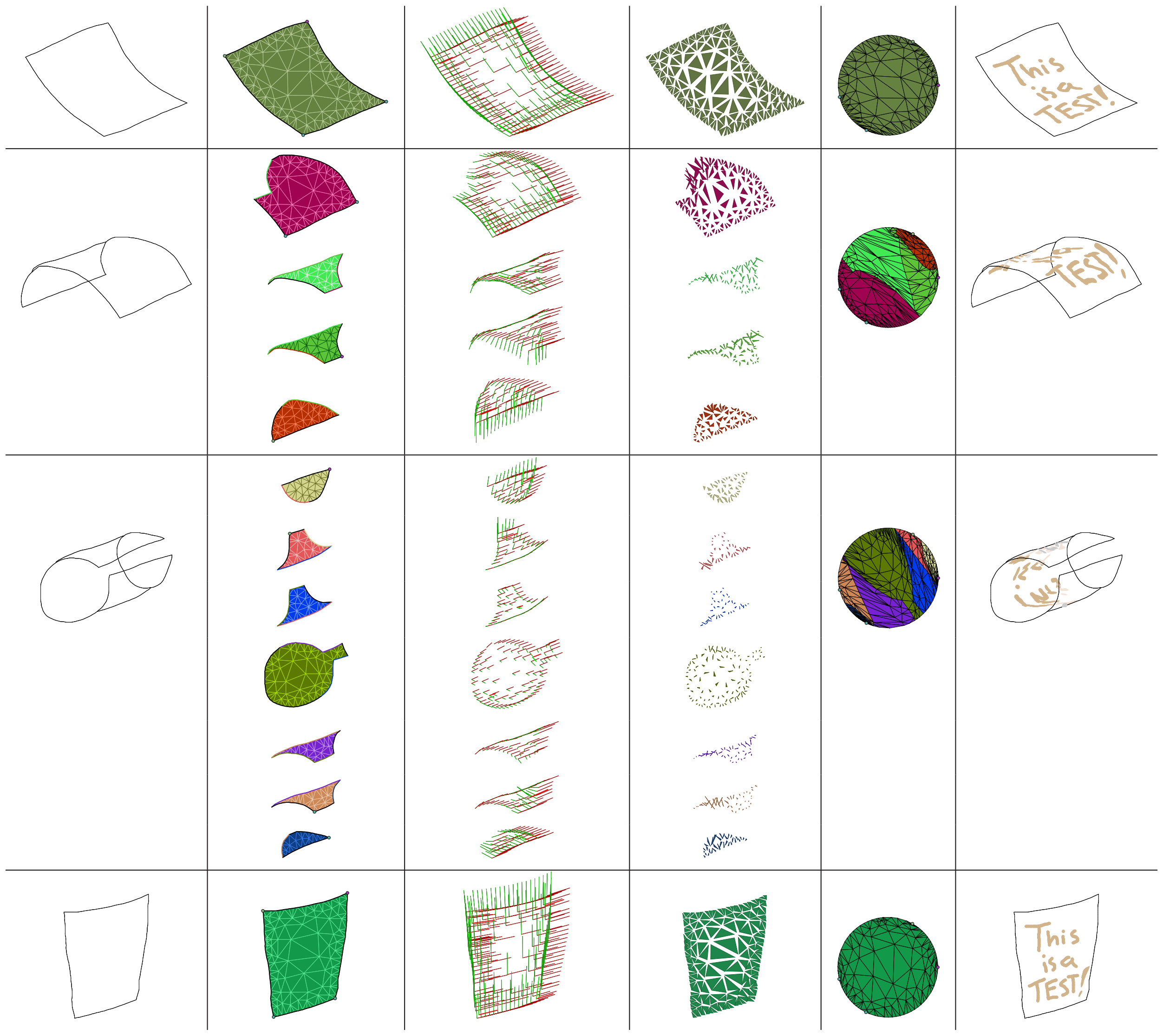


Figure 6.7: *The 3D-like texturing pipeline. From left to right: starting outlines (drawn by artist), 2.5D panels (created and glued by artist, meshed by system), estimated shape (thick lines are constraints, thin are interpolated), unwarped triangles, unwrapping to uniform texture space, and an example of texture transfer.*

Of course, before it can perform any of these tasks, a representation of the surface is required. For this, the system uses a set of 2D meshes – one per panel – which align along glued boundaries.

An illustration of these steps is provided in Figure 6.7. This will be a running example, with larger versions of each sub-figure included with each subsection.

6.6.1 Meshing

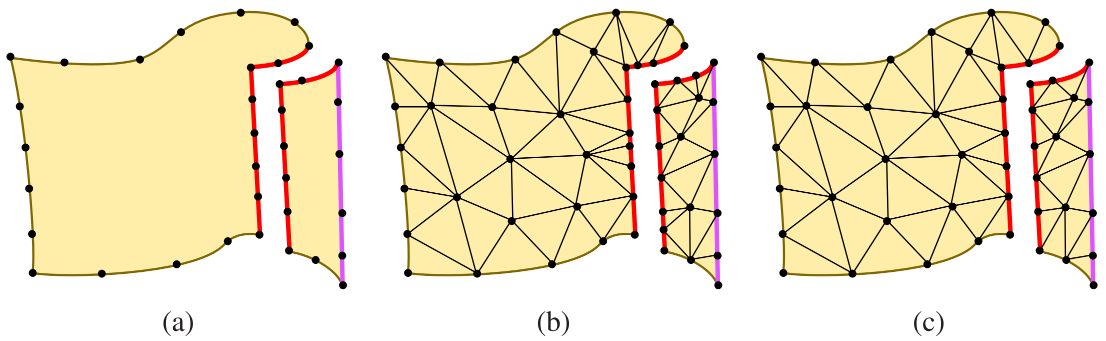


Figure 6.8: *Building meshes that agree along glued boundaries (thick, same-colored edges) by first,* **(a)***, subdividing boundaries consistently to a "fine enough" resolution; then,* **(b)***, running a mesher that may introduce points; and finally,* **(c)***, removing extra points along the edges.*

Given a set of panels and gluing rules, my method needs to mesh the panels so that any edges identified by glue are consistently subdivided. (Figure 6.9 shows this meshing method applied to four glued panels representing a curled sheet.) I will avoid calling this compatible meshing, as that name actually refers to the process of meshing two configurations of the same vertices such that the mesh is valid for both configurations [Aronov et al. 1993]. This is both a well-studied problem, and not the problem I am trying to solve here.

In order to produce good quality meshes, I use an off-the-shelf constrained Delaunay mesher from the CGAL library [CGAL]. I run the mesh generator on each panel separately. However, as the meshing algorithm may split constraints during its refinement process, this may produce meshes that don't match up along panel boundaries. I avoid this problem by splitting panel boundaries to what I deem a "fine enough" resolution, meshing, and then removing any further splits introduced by the mesher (Figure 6.8).

I represent the result of this meshing process as a list of locations and a list of triangles, where each triangle is a triple of location indices. If the same location appears in the meshes of two panels and that location is not on a glued edge between then, then it will appear twice in the list of locations – this ensures that the topology of the network of triangles accurately represents the glued panels.

For later convenience in definitions, indices of front-facing triangles (that is, triangles belonging to a front-facing panel) will appear in counter-clockwise order; back-facing triangles are stored in clockwise order.

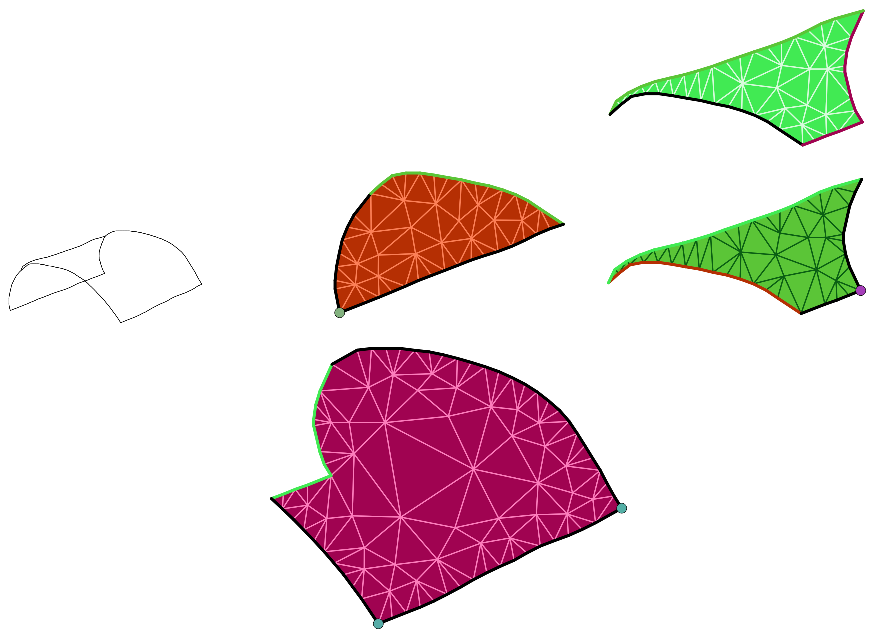


Figure 6.9: *The second frame of the folded paper example (outline,* **left***) is made of four glued panels (edge color indicates panel to which to attach). The two green panels correspond to the top and bottom layers in the area that the paper overlaps twice.*

6.6.2 Shape Estimation

The system represents object shape locally by storing, for every location, a pair of vectors (u, v) which represent the direction and magnitude of the projection of the u and v directions on the surface. These vectors are calculated by as-smooth-as-possible interpolation of user-specified surface shape constraints and by contours. Figure 6.10 shows an example from my system.

This interpolation is performed by minimizing the following objective function over triangles T and occluding contours E:

$$(U, V) \equiv \text{argmax}_{U,V} \sum_{t \in T} S(t) + M(t) + \sum_{e \in E} C(e) \tag{6.1}$$

Here S, a smoothness term, penalizes for changes in (u, v) across a triangle, M penalizes for values of u and v that don't match user-specified constraints, and C penalizes for u and v which have components normal to contours.

For the purposes of defining this objective function it will be useful to define the result of interpolating

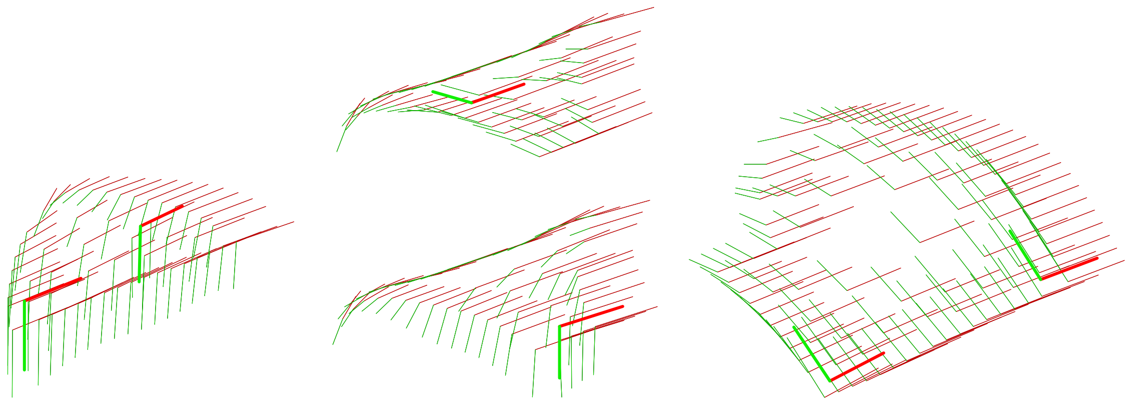


Figure 6.10: *Interpolated shape descriptors for the four panels making up a folded sheet. Thicker-lined descriptors are constraints. Interpolation continues across glued edges.*

values v_a, v_b, v_c linearly from the vertices of triangle a, b, c to point p:

$$L_{a,b,c}^{v_a,v_b,v_c}(p) \equiv \begin{aligned} & \frac{\perp(b-a)}{\perp(b-a)\cdot(c-a)}\cdot(p-a)\cdot v_c \\ + \; & \frac{\perp(c-b)}{\perp(c-b)\cdot(a-b)}\cdot(p-b)\cdot v_a \\ + \; & \frac{\perp(a-c)}{\perp(a-c)\cdot(b-c)}\cdot(p-c)\cdot v_b \end{aligned} \tag{6.2}$$

Here all vectors are columns; $a \cdot b$ is the inner product $a^T \times b$; and $\perp$ takes the perpendicular of a vector by rotating ninety degrees counterclockwise:

$$\perp \equiv \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \tag{6.3}$$

With linear interpolation in hand, defining each term of the objective function is straightforward. The smoothness term, S, just penalizes for the partial derivatives of L inside triangle t (with vertices t_1, t_2, t_3 which assume values $\{u_x, u_y, v_x, v_y\}_{\{1,2,3\}}$) with respect to position:

$$S(t) \equiv \sum_{\bullet \in \{u_x,u_y,v_x,v_y\}} \int_{\triangle_t} \left\| \frac{\partial L_{t_1,t_2,t_3}^{\bullet_1,\bullet_2,\bullet_3}(p)}{\partial p} \right\|^2 \tag{6.4}$$

Noticing that L is linear, so each of these partials is constant with respect to p, and that the denominator of each fraction in L is double the signed area of the triangle, one can simplify this expression to:

$$S(t) \equiv \sum_{\bullet \in \{u_x,u_y,v_x,v_y\}} \frac{|\text{Area}(t)|}{4\text{Area}(t)^2} \left\| \perp(t_3-t_2)\cdot\bullet_1 + \perp(t_1-t_3)\cdot\bullet_2 + \perp(t_2-t_1)\cdot\bullet_3 \right\|^2 \tag{6.5}$$

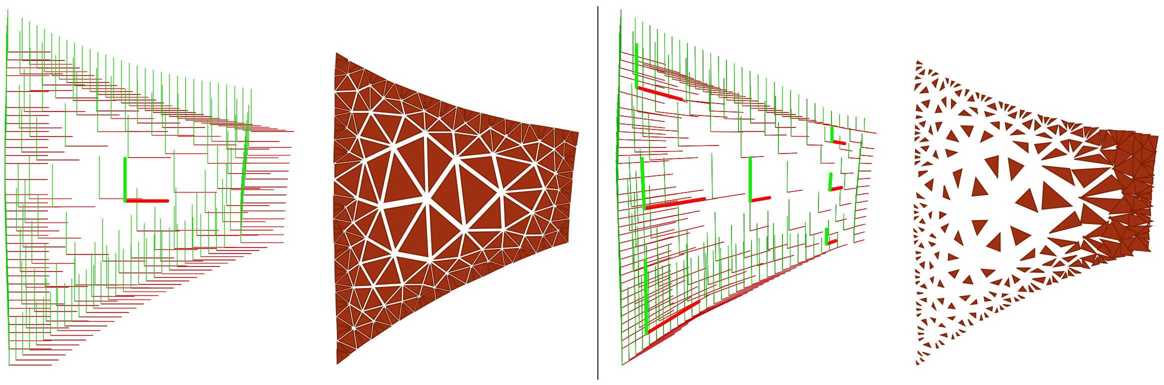


Figure 6.11: Left: *the local shape is flat, so triangles are unwarped to roughly their original size (here scaled down slightly for legibility).* **Right:** *the local shape indicates a perspective projection; triangles on the right end of the wall unwarp to relatively larger than those on the left.*

The user-matching term, $M(t)$, penalizes for differences between the user constraints, $\hat{u}, \hat{v}$ at point $\hat{p}$, and the interpolated value:

$$M(t) \equiv \sum_{(\hat{p},\hat{u},\hat{v})} \sum_{\bullet \in \{u_x,u_y,v_x,v_y\}} \left(L_{t_1,t_2,t_3}^{\bullet_1,\bullet_2,\bullet_3}(\hat{p}) - \hat{\bullet}\right)^2 \tag{6.6}$$

Where the first sum is taken over all constraints appearing inside triangle t.

Finally, the contour term $C(e)$ requires the component of u and v perpendicular to occluding contours is zero, where edge e has vertices e_1, e_2 which assume values $\{u, v\}_{\{1,2\}}$:

$$\sum_{\bullet \in u,v} \int_{\backslash} \left(\frac{\perp (e_2 - e_1)}{\|e_2 - e_1\|} \cdot (\bullet_2(1-t) + \bullet_1 t) \right)^2 \tag{6.7}$$

(Notice that while the integral is taken along the line, and thus with respect to position, I've written the interpolation from $\bullet_1$ to $\bullet_2$ with $t \in [0, 1]$ for simplicity.)

These objective terms are all quadratic and convex in U, V, so computing the optimal values can be done by a linear system solve.

6.6.3 Unwarping

Given a local shape estimate (the projection of u and v vectors) at each vertex, the system can estimate a source shape $s = (s_1, s_2, s_3)$ for each triangle (Figure 6.11). Because the objective function the system uses for unwrapping is independent of global translation or rotation of triangles, it is sufficient to choose s_1, s_2, s_3 in 2D, even though the true shape may be three- or higher-dimensional. Similarly, the "source position" corresponding to a vertex in one triangle need not match with the source position for the same vertex in another triangle.

In order to estimate an inverse to the unknown projection $\mathbf{W}$, I assume that $\mathbf{W}$ is a 2x2 matrix – this is an approximation, but a sufficient one, as this is a local estimate and need not account for global perspective effects – and then choose an inverse that aligns the interpolated u, v vectors on each triangle as closely as possible with the $[1,0]^T, [0,1]^T$ basis vectors in the source space:

$$\mathbf{W}^{-1} \equiv \operatorname{argmax}_{\mathbf{W}^{-1}} \int_{\triangle_t} \left\| \mathbf{W}^{-1}\left(L^{u_1,u_2,u_3}_{t_1,t_2,t_3}(p)\right) - \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\|^2 + \left\| \mathbf{W}^{-1}\left(L^{v_1,v_2,v_3}_{t_1,t_2,t_3}(p)\right) - \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\|^2 \tag{6.8}$$

Notice that this integral is taken over the projected triangle (not the source triangle) – the notion being that we'd like to minimize visible error.

6.6.4 Unwrapping

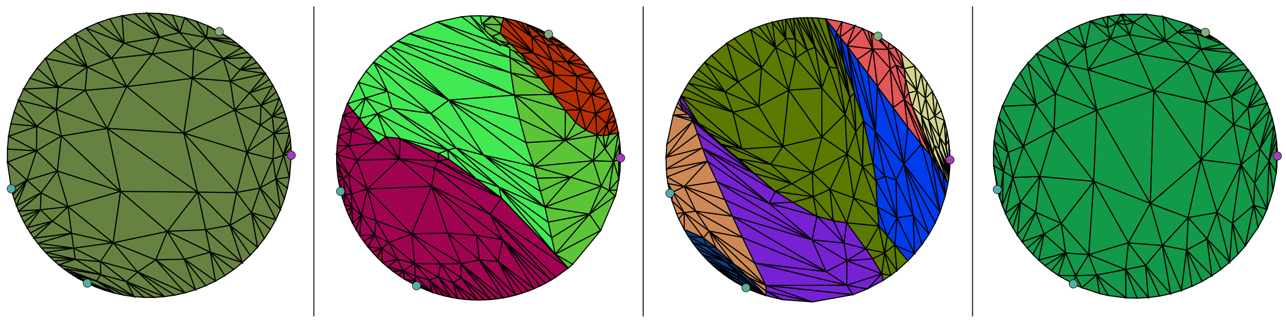


Figure 6.12: *The LSCM unwrapping of the running four-frame folded paper example. Notice the small circles around the outside of each frame which indicate artist-specified feature points.*

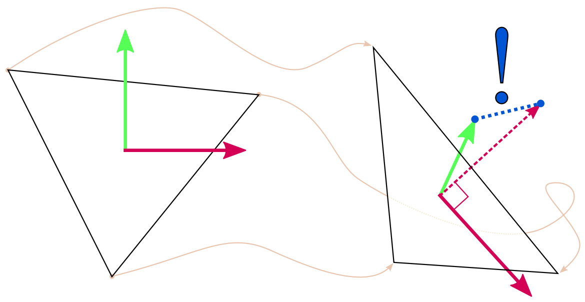


Figure 6.13: *The least-squares conformal map objective function penalizes a transformation from a source triangle* **(left)** *to a parameter space* **(right)** *for any difference between the transformed y direction (green) and the perpendicular to the transformed x direction (red).*

In order to arrive at a correspondence between the (non-embedded) unwarped meshes for each frame, some sort of consistent parameterization is required. One possibility would be to use remeshing, e.g., [Praun et al. 2001]. Instead, I take inspiration from the literature on 2D texture map generation, and make use of the least-squares conformal map (LSCM) objective function [Lévy et al. 2002] to unwrap each mesh into a circular domain in a way that respects the shape of the source triangles (Figure 6.12).

Specifically, LSCM unwrapping calculates a parameter space position p for every vertex such that the source x and y directions remain nearly orthogonal and of equal length when transformed into parameter space (Figure 6.13):

$$P \equiv \operatorname{argmax}_P \sum_{t\in T} \int_{\triangle_s} \left\| \frac{\partial L^{p_1,p_2,p_3}_{s_1,s_2,s_3}(p)}{\partial p_x} + \perp \frac{\partial L^{p_1,p_2,p_3}_{s_1,s_2,s_3}(p)}{\partial p_y} \right\|^2 \tag{6.9}$$

Since the partials are, conveniently, constant, one can re-write this as an area-weighted sum:

$$P \equiv \operatorname{argmax}_P \sum_{t\in T} |\text{Area}(\triangle_s)| \left(\left(\perp \frac{\partial L^{p_1,p_2,p_3}_{s_1,s_2,s_3}}{\partial p_x} \right) \cdot \frac{\partial L^{p_1,p_2,p_3}_{s_1,s_2,s_3}}{\partial p_y} - 1 \right)^2 \tag{6.10}$$

Notice that if no vertices are constrained there is a trivial minimizer (everything maps to zero). Therefore, one needs to pin at least two vertices when performing this unwrapping. In my system, I go further and pin the entire border to a circle. Specifically, a transformation from the border of the mesh for each frame I wish to place into correspondence to the border of the circle is computed so that:

1. feature points align,
2. unwarped edge lengths are respected as much as possible.

To do so, the system first computes the portion of the border of each frame occupied by the segment between each pair of subsequent feature points:

$$\alpha^f_{a,b} \equiv \frac{\text{len}^f(a,b)}{\text{len}^f(a,a)} \tag{6.11}$$

Here len^f computes the length along frame f's unwarped border.

The final portion of the circle's border occupied by each segment between feature points is the average of these numbers over all frames:

$$\alpha^*(a,b) \equiv \frac{1}{|F|} \sum_{f\in F} \alpha^f(a,b) \tag{6.12}$$

The system then assigns points on the boundary of each frame to positions along the circle based on the relative distance to their nearest feature points:

$$\theta_x \equiv 2\pi \left(\alpha^*(a,b) \frac{\text{len}^f(a,x)}{\text{len}^f(a,b)} + \alpha^*(0,a) \right) \tag{6.13}$$

$$p_x \equiv [\cos\theta_x, \sin\theta_x]^T \tag{6.14}$$

The system computes the unwrapped positions of non-border points by minimizing the LSCM objective function (Equation (6.10)).

6.6.5 Detail Transfer

Figure 6.14: *Once each frame is mapped to a common circle, transferring texture between them can be done by mapping from the source frame to the common parameter domain circle and back to the destination frame.*

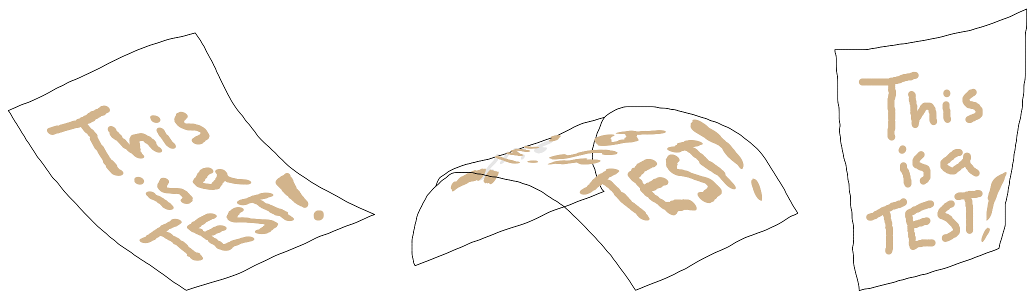


Figure 6.15: *Once each frame is mapped to a common parameter domain, any detail painted on one frame can be automatically transfered to the others.*

Texture may now be transfered between frames by mapping first to the circular common domain and then back to a target frame (see Figure 6.14 for a cartoon and Figure 6.15 for an actual result).

As my system represents paint strokes on a surface as a series of ellipses, it also scales and shears these ellipses when mapping from frame to frame.

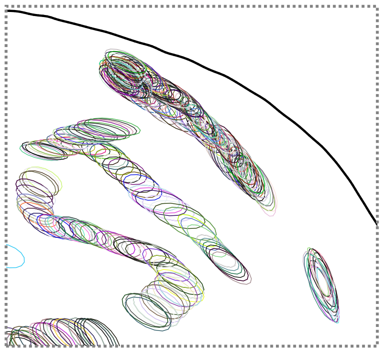


Figure 6.16: *An enlargement showing the borders of the ellipses which make up the paint strokes in the middle panel of Figure 6.15.*

6.7 Discussion

My texturing approach can handle squashing and stretching objects (Figure 6.17), as well as deformations of rigid objects such as a fluttering page (Figure 6.15), and camera transformations (Figure 6.1).

However, to be fair, it does not handle any of these transformations particularly well; it is often hard to understand the current texturing and what additional constraints are needed to achieve the desired texturing. While I believe the underlying idea of the method (direct texture coordinate generation on 2D patches) is solid, I think future investigations in this area should focus on two areas for improvement. First, I discuss how one might improve the interface experience. The axis-pair gizmo I introduce for local stretch specification may well be too local – it's often difficult to select consistent axis pairs across large planar or nearly-planar regions. Contrast this to a local shape specification based on, say, drawing regions and associating normals, or painting normals and using a separate method altogether to define stretch (both seem like interesting potential explorations). Additionally, there is no interface given to adjust the generated texture coordinates directly (one must instead tweak surface shapes and hope). This lack of interactive manipulation runs counter to one of the themes of my thesis so far (simple, direct, local tools), and can be very frustrating in practice – sometimes the automatic coordinates are almost correct, but that last little bit is nearly impossible to extract. Compounding these problems is a lack of good surface shape or correspondence visualization (something that runs contrary to the notion of user feedback required by my thesis).

The second area future work should target for improvement is the actual texture coordinate generation. Though the underlying pipeline – unwarp and then align/unwrap – seems sound, there are problems in both stages. Unwarping happens at a triangle level, and this often leads to wildly inconsistent edge lengths in neighboring triangles. This causes some algorithmic inconvenience[3], and seems entirely avoidable by using a slightly less local unwarping operator. Warping aside, unwrapping works well for small and relatively uniform regions, but begins to fail on large regions with lots of normal variation – this is because there is no method (automatic or – as mentioned above – user-driven) to align interior points when unwrapping.

[3]These inconsistencies seem to lead the least-squares conformal maps technique into occasional – small – self-overlap situations.

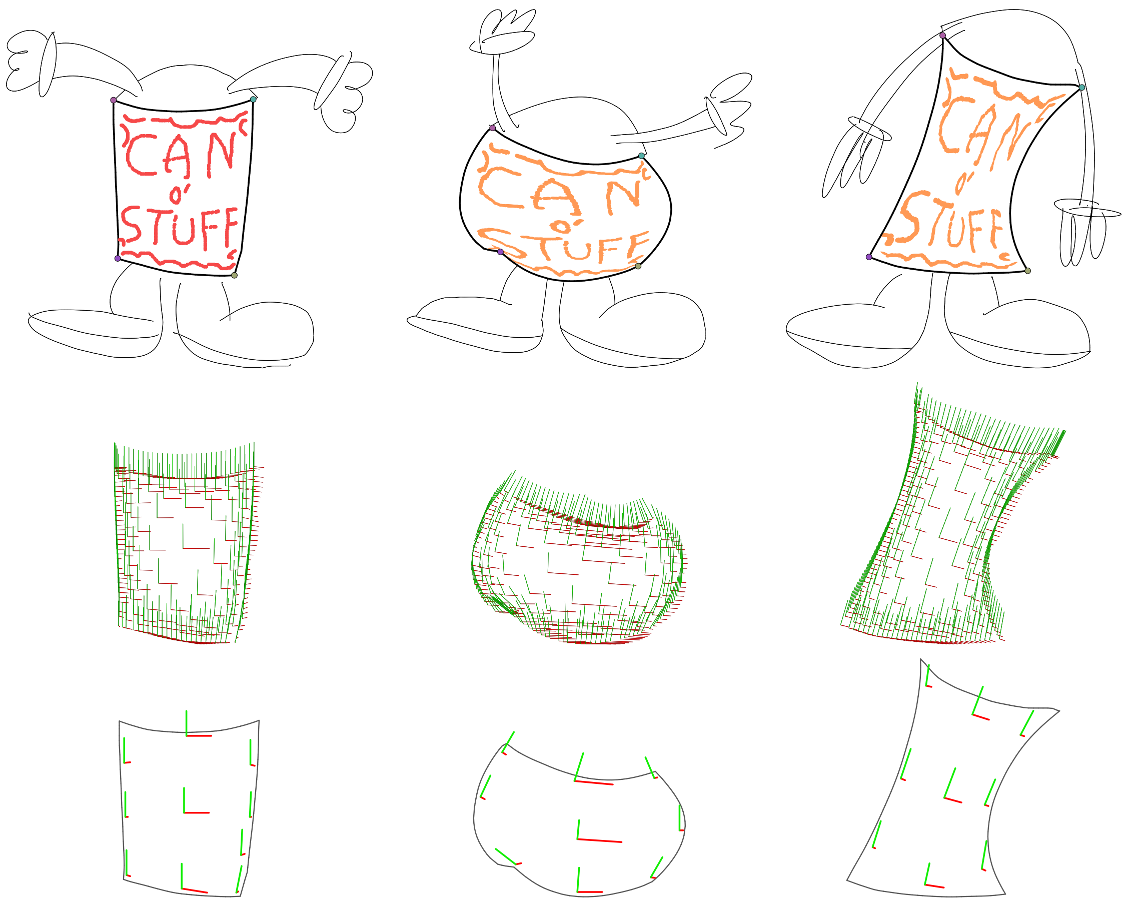


Figure 6.17: *A squishy can. My system does not require a global (or consistent-across-frames) model of the surface to perform texture transfer. The local shape model (middle row) is specified by sparse constraints (bottom row).*

Despite these problems, I have presented in this chapter the beginnings of a method that uses local and intuitive manipulation to specify local surface shape while preserving global shape ambiguity, and performs surface detail transfer automatically with this information.

Chapter 7

Concluding Remarks

In this document, I presented four new image and animation editing systems, all motivated by the same core premise:

> New image editing tools may be created, and existing ones improved, by allowing interactive edits of perception-motivated local features.

In each system, I sought local features appropriate to a given task, and built algorithms to interpret these features.

This approach to exploring the space of possible graphics tools – picking a relevant local feature and building tools to manipulate it – has been a fruitful one. At best, the resulting systems offer easier ways to perform previously cumbersome edits; at worst, the programs are merely interesting. And, on rare occasions, I find myself making art in ways that no one has ever made art before.

Rhapsodizing aside, there are more projects in this space to tackle. Two examples:

- **Texture.** Though off-the-shelf methods for synthesizing and recognizing coherent texture exist, manipulating surface texture is still quite cumbersome for users. This may be because tool designers have not found perceptually relevant texture features.
- **Depth.** With the recent resurgence of techniques for presenting different views of a scene to each of a viewer's eyes, the creation of proper depth cues has become important to industry. However, stereoscopic disparity is just one of many depth cues – and is, in fact, not particularly stronger than the others. A system could be designed to exploit this: give users control over the perception of depth, and factor this into several different cues (including disparity).

But these are small things. On a larger scale, and in the present historical context, this thesis suggests two important directions for the future:

- **Forget locality.** I chose local features to explore because the world still uses primarily local input devices (mice, styluses), and local features appear easier to work with. But with the boom in multi-touch and camera input devices, locality may no longer make sense.

- **Figure out perception.** Perceptual psychology and vision researchers continue to push out the frontiers of what is known about the entire human vision pipeline, both from the bottom (using functional MRI to explore layers of neural filters and encoders) and from the top (perceptual studies and generation of isomorphic stimuli). These new results can, and should, inform the design of new graphics tools.

In closing, I have a simple observation. When building a new tool, it is much easier to characterize it in terms of sufficiency (“the user can reach all states”) than in terms of relevance (“the tool allows the user to edit the right thing”). However, the latter is much more important in practice.

Bibliography

[Adelson and Wang 1994] ADELSON, E. H., AND WANG, J. Y. A. 1994. Representing moving images with layers. *IEEE Transactions on Image Processing 3*, 625–638.

[Adobe 2008] ADOBE, 2008. Photoshop cs4. http://www.adobe.com/products/photoshop.

[Agarwala 2007] AGARWALA, A. 2007. Efficient gradient-domain compositing using quadtrees. *ACM Transactions on Graphics 26*, 3.

[Agrawal and Raskar 2007] AGRAWAL, A., AND RASKAR, R., 2007. Gradient domain manipulation techniques in vision and graphics. ICCV 2007 Course.

[Apple 1999] APPLE, 1999. Final Cut Pro. http://www.apple.com/finalcutstudio/finalcutpro/.

[Aronov et al. 1993] ARONOV, B., SEIDEL, R., AND SOUVAINE, D. 1993. On compatible triangulations of simple polygons. *Computational Geometry: Theory and Applications 3*, 27–35.

[Asente et al. 2007] ASENTE, P., SCHUSTER, M., AND PETTIT, T. 2007. Dynamic planar map illustration. *ACM Transactions on Graphics 26*, 3, 30.

[Baudelaire and Gangnet 1989] BAUDELAIRE, P., AND GANGNET, M. 1989. Planar maps: An interaction paradigm for graphic design. *SIGCHI Bull. 20*, SI, 313–318.

[Bolz et al. 2003] BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖODER, P. 2003. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics 22*, 3, 917–924.

[Brainard and Wandell 1986] BRAINARD, D. H., AND WANDELL, B. A. 1986. Analysis of the retinex theory of color vision. *Journal of the Optical Society of America A: Optics, Image Science, and Vision 3* (Oct.), 1651–1661.

[Brandt 1976] BRANDT, A., 1976. Multi-level adaptive techniques. IBM Research Report RC6026.

[Burt and Adelson 1983] BURT, P. J., AND ADELSON, E. H. 1983. A multiresolution spline with application to image mosaics. *ACM Transactions on Graphics 2*, 4, 217–236.

[CGAL] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

[Cordier and Seo 2007] CORDIER, F., AND SEO, H. 2007. Free-form sketching of self-occluding objects. *IEEE Comput. Graph. Appl. 27*, 1, 50–59.

[Corrêa et al. 1998] CORRÊA, W. T., JENSEN, R. J., THAYER, C. E., AND FINKELSTEIN, A. 1998. Texture mapping for cel animation. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 435–446.

[Curtis et al. 1997] CURTIS, C. J., ANDERSON, S. E., SEIMS, J. E., FLEISCHER, K. W., AND SALESIN, D. H. 1997. Computer-generated watercolor. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 421–430.

[Debevec et al. 1996] DEBEVEC, P. E., TAYLOR, C. J., AND MALIK, J. 1996. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *Proceedings of SIGGRAPH 96*, ACM, New York, NY, USA, 11–20.

[Duff 1985] DUFF, T. 1985. Compositing 3-D rendered images. *Computer Graphics (Proceedings of SIGGRAPH 85) 19*, 3, 41–44.

[Eastman and Wooten 1974] EASTMAN, J. F., AND WOOTEN, D. R. 1974. A general purpose, expandable processor for real-time computer graphics. In *SIGGRAPH '74: Proceedings of the 1st annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 30–30.

[Elder and Goldberg 2001] ELDER, J. H., AND GOLDBERG, R. M. 2001. Image editing in the contour domain. *IEEE Transactions on Pattern Analysis and Machine Intelligence 23*, 3, 291–296.

[Everitt 2001] EVERITT, C. 2001. Introduction to interactive order-independent transparency. Tech. rep., NVIDIA.

[Fang and Hart 2004] FANG, H., AND HART, J. C. 2004. Textureshop: Texture synthesis as a photograph editing tool. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, ACM, New York, NY, USA, 354–359.

[Fatahalian et al. 2010] FATAHALIAN, K., BOULOS, S., HEGARTY, J., AKELEY, K., MARK, W. R., MORETON, H., AND HANRAHAN, P. 2010. Reducing shading on gpus using quad-fragment merging. In *SIGGRAPH '10: ACM SIGGRAPH 2010 papers*, ACM, New York, NY, USA, 1–8.

[Fattal et al. 2002] FATTAL, R., LISCHINSKI, D., AND WERMAN, M. 2002. Gradient domain high dynamic range compression. *ACM Transactions on Graphics 21*, 3, 249–256.

[Fedorenko 1962] FEDORENKO, R. P. 1962. A relaxation method for solving elliptic difference equations. *USSR COmputation math and Mathematical Physics*, 4, 1092–1096.

[Finlayson et al. 2002] FINLAYSON, G., HORDLEY, S., AND DREW, M. 2002. Removing shadows from images. In *ECCV 2002: European Conference on Computer Vision*.

[Fiore and Reeth 2002] FIORE, F. D., AND REETH, F. V. 2002. Employing approximate 3D models to enrich traditional computer assisted animation. *Computer Animation 0*, 183.

[Fiore et al. 2001] FIORE, F. D., SCHAEKEN, P., ELENS, K., AND REETH, F. V. 2001. Automatic in-betweening in computer assisted animation by exploiting 2.5D modelling techniques. In *In Proceedings of Computer Animation (CA2001)*.

[Fractal Design 1992] FRACTAL DESIGN, 1992. Painter. http://www.corel.com/painter.

[Fu et al. 2010] FU, C.-W., XIA, J., AND HE, Y. 2010. Layerpaint: a multi-layer interactive 3D painting interface. In *CHI '10: Proceedings of the 28th international conference on human factors in computing systems*, ACM, New York, NY, USA, 811–820.

[Galyean and Hughes 1991] GALYEAN, T. A., AND HUGHES, J. F. 1991. Sculpting: An interactive volumetric modeling technique. *SIGGRAPH Comput. Graph. 25*, 4, 267–274.

[Goodnight et al. 2003] GOODNIGHT, N., WOOLLEY, C., LEWIN, G., LUEBKE, D., AND HUMPHREYS, G. 2003. A multigrid solver for boundary value problems using programmable graphics hardware. In *HWWS '03*, 102–111.

[Greene 1985] GREENE, R. 1985. The drawing prism: A versatile graphic input device. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 103–110.

[Hickman 1989] HICKMAN, C., 1989. Kid Pix. http://pixelpoppin.com/kidpix.

[Igarashi et al. 2005] IGARASHI, T., MOSCOVICH, T., AND HUGHES, J. F. 2005. As-rigid-as-possible shape manipulation. *ACM Transactions on Graphics 24*, 3, 1134–1141.

[Jeschke et al. 2009] JESCHKE, S., CLINE, D., AND WONKA, P. 2009. A gpu laplacian solver for diffusion curves and poisson image editing. In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, ACM, New York, NY, USA, 1–8.

[Karpenko and Hughes 2006] KARPENKO, O. A., AND HUGHES, J. F. 2006. Smoothsketch: 3D free-form shapes from complex sketches. *ACM Transactions on Graphics 25*, 3 (July), 589–598.

[Kazhdan and Hoppe 2008] KAZHDAN, M., AND HOPPE, H. 2008. Streaming multigrid for gradient-domain operations on large images. *ACM Transactions on Graphics 27*, 3.

[Keefe et al. 2001] KEEFE, D., ACEVEDO, D., MOSCOVICH, T., LAIDLAW, D. H., AND LAVIOLA, J. 2001. CavePainting: A fully immersive 3D artistic medium and interactive experience. In *Proceedings of ACM Symposium on Interactive 3D Graphics*, 85–93.

[Kimball and Mattis 1995–2009] KIMBALL, S., AND MATTIS, P., 1995–2009. The GNU Image Manipulation Program. http://www.gimp.org.

[Koenderink et al. 1992] KOENDERINK, J. J., VAN DOORN, A. J., AND KAPPERS, A. M. L. 1992. Surface perception in pictures. *Perception and Psychophysics 52*, 5, 487–496.

[Koenderink et al. 2001] KOENDERINK, J. J., VAN DOORN, A. J., KAPPERS, A. M. L., AND TODD, J. T. 2001. Ambiguity and the 'mental eye' in pictorial relief. *Perception 30*, 4, 431 – 448.

[Koutis 2007] KOUTIS, I., 2007. Combinatorial and algebraic tools for optimal multilevel algorithms. PhD Thesis, CMU-CS-07-131.

[Land and McCann 1971] LAND, E. H., AND MCCANN, J. J. 1971. Lightness and retinex theory. *Journal of the Optical Society of America (1917-1983) 61* (Jan.), 1–11.

[Levin et al. 2003] LEVIN, A., ZOMET, A., PELEG, S., AND WEISS, Y., 2003. Seamless image stitching in the gradient domain. Hebrew University Tech Report 2003-82.

[Lévy et al. 2002] LÉVY, B., PETITJEAN, S., RAY, N., AND MAILLOT, J. 2002. Least squares conformal maps for automatic texture atlas generation. *ACM Trans. Graph. 21*, 3, 362–371.

[Lischinski et al. 2006] LISCHINSKI, D., FARBMAN, Z., UYTTENDAELE, M., AND SZELISKI, R. 2006. Interactive local adjustment of tonal values. *ACM Transactions on Graphics 25*, 3, 646–653.

[Markosian 2000] MARKOSIAN, L. 2000. *Art-based modeling and rendering for computer graphics*. PhD thesis, Brown University, Providence, RI, USA.

[McDonnell et al. 2001] MCDONNELL, K. T., QIN, H., AND WLODARCZYK, R. A. 2001. Virtual clay: A real-time sculpting system with haptic toolkits. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, ACM, New York, NY, USA, 179–190.

[Mediachance 2001–2009] MEDIACHANCE, 2001–2009. Real-Draw PRO push-back tool. http://www.mediachance.com/realdraw/help/pushback.htm.

[Nitzberg and Mumford 1990] NITZBERG, M., AND MUMFORD, D. 1990. The 2.1-d sketch. *Computer Vision, 1990. Proceedings, Third International Conference on* (Dec), 138–144.

[Orzan et al. 2008] ORZAN, A., BOUSSEAU, A., WINNEMOELLER, H., BARLA, P., JOËLLE, AND SALESIN, D. 2008. Diffusion curves: A vector representation for smooth shaded images. *ACM Transactions on Graphics 27*, 3.

[Pérez et al. 2003] PÉREZ, P., GANGNET, M., AND BLAKE, A. 2003. Poisson image editing. *ACM Transactions on Graphics 22*, 3, 313–318.

[Pixologic 2004] PIXOLOGIC, 2004. Zbrush. http://www.pixologic.com/zbrush.

[Porter and Duff 1984] PORTER, T., AND DUFF, T. 1984. Compositing digital images. *Computer Graphics (Proceedings of SIGGRAPH 84) 18*, 3, 253–259.

[Praun et al. 2001] PRAUN, E., SWELDENS, W., AND SCHRÖDER, P. 2001. Consistent mesh parameterizations. In *SIGGRAPH '01: Proceedings of the 28th annual conference on computer graphics and interactive techniques*, ACM, New York, NY, USA, 179–184.

[Press et al. 1992] PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 1992. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, ch. 19.6, 871–888.

[Rademacher 1999] RADEMACHER, P. 1999. View-dependent geometry. In *SIGGRAPH '99: Proceedings of the 26th annual conference on computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 439–446.

[Rivers et al. 2010a] RIVERS, A., DURAND, F., AND IGARASHI, T. 2010. 3D modeling with silhouettes. In *SIGGRAPH '10: ACM SIGGRAPH 2010 papers*, ACM, New York, NY, USA, 1–8.

[Rivers et al. 2010b] RIVERS, A., IGARASHI, T., AND DURAND, F. 2010. 2.5D cartoon models. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010)*.

[Roberts 2001] ROBERTS, A. J. 2001. Simple and fast multigrid solution of Poisson's equation using diagonally oriented grids. *ANZIAM J. 43*, E (July), E1–E36.

[Schmid et al. 2010] SCHMID, J., SUMNER, R. W., BOWLES, H., AND GROSS, M. 2010. Programmable motion effects. In *SIGGRAPH '10: ACM SIGGRAPH 2010 papers*, ACM, New York, NY, USA, 1–9.

[Schödl et al. 2000] SCHÖDL, A., SZELISKI, R., SALESIN, D. H., AND ESSA, I. 2000. Video textures. In *Proceedings of ACM SIGGRAPH 2000*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 489–498.

[Shoup 2001] SHOUP, R. 2001. Superpaint: An early frame buffer graphics system. *IEEE Annals of the History of Computing 23*, 32–37.

[Smith 1974] SMITH, L. B. 1974. An example of a pragmatic approach to portable interactive graphics. In *SIGGRAPH '74: Proceedings of the 1st annual conference on computer graphics and interactive techniques*, ACM, New York, NY, USA, 18–18.

[Smith 1995] SMITH, A. R. 1995. Alpha and the history of digital compositing. In *Microsoft Technical Memo #7*.

[Snyder and Lengyel 1998] SNYDER, J., AND LENGYEL, J. 1998. Visibility sorting and compositing without splitting for image layer decompositions. In *Proceedings of SIGGRAPH 98*, ACM, New York, NY, USA, 219–230.

[Staudhammer and Ogden 1974] STAUDHAMMER, J., AND OGDEN, D. J. 1974. Computer graphics for half-tone three-dimentional object images. In *SIGGRAPH '74: Proceedings of the*

1st annual conference on Computer graphics and interactive techniques, ACM, New York, NY, USA, 36–36.

[Strassmann 1986] STRASSMANN, S. 1986. Hairy brushes. *SIGGRAPH Comput. Graph. 20*, 4, 225–232.

[Sutherland 1963] SUTHERLAND, I. E., 1963. Sketchpad: A man-machine graphical communication system. PhD Thesis.

[Sýkora et al. 2010] SÝKORA, D., SEDLÁČEK, D., JINCHAO, S., DINGLIANA, J., AND COLLINS, S. 2010. Adding depth to cartoons using sparse depth (in)equalities. *Computer Graphics Forum 29*, 2, 615–623.

[Szeliski 2006] SZELISKI, R. 2006. Locally adapted hierarchical basis preconditioning. *ACM Transactions on Graphics 25*, 3, 1135–1143.

[Tumblin et al. 2005] TUMBLIN, J., AGRAWAL, A., AND RASKAR, R. 2005. Why I want a gradient camera. In *Proceedings of IEEE CVPR 2005*, vol. 1, 103–110.

[Werner 1935] WERNER, H. 1935. Studies on contour: I. Qualitative analyses. *The American Journal of Psychology 47*, 1 (Jan.), 40–64.

[Wiley 2006] WILEY, K. 2006. *Druid: Representation of Interwoven Surfaces in 2 1/2 D Drawing*. PhD thesis, University of New Mexico.

[Williams 1997] WILLIAMS, L. R. 1997. Topological reconstruction of a smooth manifold-solid from its occluding contour. *International Journal of Computer Vision 23*, 1, 93–108.

[Zhang et al. 2001] ZHANG, L., DUGAS-PHOCION, G., SAMSON, J.-S., AND SEITZ, S. M. 2001. Single view modeling of free-form scenes. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on 1*, 990.