A Sprite Theory of Image Computing

Alvy Ray Smith

Hume on spatial extension (and alpha) in 1739:

> The table before me is alone sufficient by its view to give me the idea of extension. This idea, then, is borrow'd from, and represents some impression, which this moment appears to the senses. But my senses convey to me only the impressions of colour'd points, dispos'd in a certain manner. If the eye is sensible of any thing farther, I desire it may be pointed out to me. But if it be impossible to shew any thing farther, we may conclude with certainty, that the idea of extension is nothing but a copy of these colour'd points, and of the manner of their appearance.
>
> Suppose that in the extended object, or composition of colour'd points, from which we first receiv'd the idea of extension, the points were of a purple colour; it follows, that in every repetition of that idea we wou'd not only place the points in the same order with respect to each other, but also bestow on them that precise colour, with which alone we are acquainted. But afterwards having experience of the other colours of violet, green, red, white, black, and of all the different compositions of these, and finding a resemblance in the disposition of colour'd points, of which they are compos'd, we omit the peculiarities of colour, as far as possible, and found an abstract idea merely on that disposition of points, or manner of appearance, in which they agree.
>
> *David Hume*
> *A Treatise of Human Nature*
> *Book I, Part II, Section III*

Published by
ARS LONGA
publications imprint of
Alvy Ray Smith PhD
Seattle WA
<alvyray.com>

Bibliographic Reference:
Alvy Ray Smith, *A Sprite Theory of Image Computing* (Seattle: Ars Longa, 2009–2010).

TABLE OF CONTENTS

TABLE OF ILLUSTRATIONS

INTRODUCTION

I gather here ideas scattered over a dozen or so technical memos written over many years at several institutions into a single general theory of image computing, which I believe deserves wider appreciation. The theory springs from two fundamental notions, the first seemingly obvious, the second not so much: (1) sample-based computer picturing differs completely from geometry-based picturing—or, what is essentially the same, the discrete realm is disjoint from the continuous; and (2) the concept of premultiplied alpha frees the imaging world—the sample-based realm—from rectilinearity. In fact, the word "sprite" means, more-or-less, "shaped image" or "non-rectilinear image."

Strong consequences of the theory are: (1) overthrow of the tyranny of the rectangle in imaging via the notion of sprites; (2) a careful separation of the discrete domain from the continuous, disallowing for example rectilinear arrays thought of as rectangles, descriptions of pixels as little squares or voxels as little cubes, real coordinate systems for images, and a host of similar confusions that have plagued image computing for decades;[1] (3) a sampling-theoretical, rigorous method for going back and forth between the two domains, banishing the box reconstruction filter or at least trumpeting its general inadequacy; (4) separation of creative space from display space for images, a simple but profound idea borrowed directly from the continuous world into the discrete world; and (5) a pathway for integration of the discrete with the continuous worlds—completely separated at present, with distinct applications based on distinct interfaces—with at least a common user interaction being a consequence.

Most images are now digital, so there is no longer any reason that an image be assumed rectilinear, despite the fact that the world's most wide-

[1] For example, I avoid ever using the word "rectangular" (a geometric term) when talking about "rectilinear" (a sampling term) arrays—i.e., those arranged in discrete rows and columns. Here is another: geometric objects are "clipped," but sprites are "cropped." There will be many other instances of almost maniacal avoidance of continuous terms in discrete contexts to avoid the many confusions which currently exist between the realms.

ly used image editing application does exactly that. The mechanism for breaking the rectangle's stranglehold is the *sprite*, or shaped image—a phrasing I hope you soon find redundant—its shape being defined by its *alpha* channel, which specifies those points occupied by the sprite. The alpha channel is intrinsic to the sprite in a manner to be described, not just extra information. The other channels contain (at least) the sprite's color—often a red channel, a green channel, and a blue channel in a typical configuration. Since the alpha channel remains defined despite what colors might be substituted into these other channels, it realizes the extension, Hume's term, of the object or objects represented by the sprite. In this rephrasing of the problem, the old-fashioned rectilinear image is just a special case of the sprite, being defined on a rectilinear array of sample points with alpha equal one (that is, opaque) at each sample.

Although much of what I have to say is couched in terms of 2-dimensional images, it shall be obvious that all the notions expressed readily generalize to 3-dimensional images—so-called volumetric imaging—and to temporal sequences of 2-dimensional images such as movies or videos,[2] even to temporal sequences of 3-dimensional images,[3] and so forth. Overthrowing the tyranny of the rectangle in 2-dimensional imaging generalizes to overthrowing the tyranny of cube (or rectangular prism) in shaped volumes and sampled flows.

Digital images, and hence sprites, are famously constituted of pixels—in the 2-dimensional case anyway. In the 3-dimensional generalization to volumetric imaging the pixel is often called a voxel. A common misperception—that a pixel is a little square, or that a voxel is a little cube—is so pernicious that I devoted a paper to the topic many years ago. Despite its signs of age—examples using old-fashioned analog video, for example—I attach the paper as appendix A essentially unchanged, as I don't think I could word it better now. The important point is: In all cases of digital imaging, and in all dimensions, the constitutive element—whether it be called a pixel, or voxel, or whatever—is a sample, and classic sampling theory is the underlying mathematics. There is no geometry.

[2] Often termed (2+1)-dimensional objects.
[3] (3+1)-dimensional objects.

SPRITE THEORY

Classic plane and solid geometry underlies the other major branch of computer-mediated picturing, that of geometry-based applications in which the triangle, square, circle, polygons, splines, cylinders, ellipsoids, patches, etc., are the constitutents. There are no pixels in this world, only geometry. It is the *display* of the abstract geometry that invokes sampling and pixels, but display is a process separate from the creation of the abstract geometry-based picture to be displayed—a point I shall return to repeatedly. Geometry-based applications assume a continuous model of the universe. Sampling-based applications assume a discrete model.

The sprite theory is opposed to the "monolithic" theory, currently the most widely used theory of image computing. The monolithic theory is characterized by the resulting image always being rectilinear—that is, restricted to the bounding shape of a rectangle and opaque everywhere. The popularity of this restricted model reflects the dominance of Adobe PHOTOSHOP which employs it. In fact, I will sometimes refer to the monolithic model as the PHOTOSHOP model.

The basic monolithic theory idea is that one is given a single (rectilinear) image, to which operations are applied to generate a final (rectilinear) image. The basic sprite theory idea is a collection of free floating sprites (shaped images), typically but not necessarily overlapping, that are arranged to form a composition of sprites (a shaped image), where operations are applied to each sprite in the collection, or to groupings of the sprites. The sprite model reduces to the monolithic model if one of the sprites is made rectilinear and designated the "background" (an unnecessary distinction in sprite theory) which is forced to the back of the collection of sprites, and all other sprites are made to fall within the bounds of that background image, perhaps by cropping to it. The notion of "layers" in PHOTOSHOP is the monolithic attempt at accomplishing awkwardly what happens naturally in the sprite model. So the monolithic model is a special case of the sprite model.

The monolithic model is not consistent with geometric models of computer picturing. As a consequence the user must learn two user interface paradigms, one for geometry-based applications (such as Adobe ILLUSTRATOR) and another, quite different one for pixel-based applica-

tions (such as Adobe PHOTOSHOP, of course). This distinction is of long standing, going back, for example, to Apple MACDRAW and Apple MACPAINT, respectively.

The sprite theory of image computing is elegantly consistent with geometric models of computer picturing in that the natural user interface employed by geometry-based picturing applications is the natural interface suggested by the sprite theory for sampling-based picturing applications. I will use this consistency as the path to convergence.

Considering the beauty of the idea of the single user interface, it is surprising to me that the sprite theory continues to be insufficiently recognized. This lecture in an attempt to correct that situation. As discussed next, perhaps computation power has finally reached the point where the idea can finally take solid root.

A drawback of sprite-based applications has been the requirement of the extra channel, the alpha channel. Furthermore, since the identity of sprites is typically maintained as long as possible in a sprite-based application, the memory devouring nature of sprites was further increased. But the seemingly endless amount of almost free memory that is becoming available because of the exponential decrease in element size described by Moore's "law" is alleviating the large-memory aspect of sprite-based imaging. Moore's law promises us an order-of-magnitude improvement about every five years, with a consequent necessity to change our mode of thinking. I suggest that we can now cease thinking about the "weight" of sprites and simply start using them as freely as we use, say, triangles. This was a motivation, in fact, for selecting the lightweight word "sprite."

I have worked out the sprite theory over about 30 years, starting from the 1977 invention of the alpha channel, through definition of a language, ICEMAN, for working formally with sprites, to implementation in a prototype application turned into a product, called Altamira COMPOSER. This lecture is informed by each of these steps. Many colleagues have helped along the way, as I shall acknowledge at the appropriate places. The next section details the history of the sprite theory and the origins of its terms.

SPRITE THEORY

ORIGINS

Invention of Alpha

My partner of many years, Ed Catmull, and I invented the notion of the *integral alpha* in the 1970s at the New York Institute of Technology (NYIT, or New York Tech). This is the notion that opacity (or, interchangeably, transparency) of an image is as fundamental as its color and should therefore be included as part of the image, not as a secondary accompaniment. To be very clear, we did not invent digital mattes (non-integral alpha) or digital compositing. These were obvious digital adaptations of known analog techniques in the filmmaking world. We invented the notion of the alpha channel as a fundamental component of an image. We coined the term "alpha" for the new channel, and called the expanded pixel an "RGBA" pixel—for Red, Green, Blue, and Alpha. RGBA images—composed, of course, of RGBA pixels—became fundamental to all work done by our team from that point forward, including Lucasfilm and Pixar (which Ed and I cofounded).

Hundreds of thousands, millions surely, of images have been created in the last three decades with (integral) alpha channels. Many films have been made using them—for example, all those of Lucasfilm and its special effects division, Industrial Light & Magic, made after 1982 with digital elements, all those of Pixar after 1986 when Pixar was founded, and all Disney animated films after 1990 when Pixar implemented the Computer Animation Production System (CAPS) for Disney, and certainly after the purchase of Pixar by Disney in 2006.

It is not hard to understand why no one had leapt to the admittedly simple concept of the integral alpha before Ed and I did. At the time, the mid 1970s, memory was very expensive. Our first video framebuffer—which is what we called an 8-bit graphics card then (before high-definition video) if you can imagine a graphics card the size of a refrigerator—with 640×480×8 bits, cost $80,000, and the next five cost $60,000 each. So an RGBA framebuffer—four of these video framebuf-

fers ganged together—cost $260,000 in 1975 dollars.[4] It was nontrivial in those days to increase memory usage by 25%. And we were the only facility in the world that had a 24-bit or 32-bit framebuffer, and one of only three or four places that had even a lowly 8-bit framebuffer (ignoring the military intelligence community, because we have never known what they had of course).

I remember clearly the moment of the invention of alpha. Ed was working on his new hidden surface algorithm for a SIGGRAPH paper submission, published eventually as (Catmull 1978).[5] He tested his technique by generating images of objects over different backgrounds. I was working with him to make these pictures since I knew where the interesting background images were stored in our file system. I would position an image in the framebuffer that he would then render over, using his new technique. The compositing would happen as the rendering occurred. This was tedious because each different background required a new rendering, then an excrutiatingly slow process. Ed mentioned that it certainly would make life easier if, instead of re-rendering the same image over different backgrounds, he rendered the opacity information *once* with the color information at each pixel, put this into a file somehow, and *then* the image in the file could be composited over different backgrounds without re-rendering. Immediately I told him that this would be easy.

I could say this confidently because I had written the image file save and restore programs that we used.[6] I already had versions for saving and restoring 8-bit and 24-bit images, and I knew exactly how to write a version that would save and restore 32-bit images. I started right then and by the next morning had the full package, complete with Unix-style ma-

[4] About $1 million in 2009 dollars! *The Inflation Counter*, <www.westegg.com/inflation/>, based on the Consumer Price Index, converted $260,000 in 1975 to $991,663 in 2007. Of course, one can now buy an entire laptop computer (or smaller) with full-color high-resolution graphics for under $500. The graphics "card" is just assumed anymore and is probably just a chip on a circuit board with the rest of the computer.

[5] See bibliography for source details, and for an explanation of SIGGRAPH, should it be an unfamiliar term.

[6] I learned my imaging basics from Dick Shoup at the Xerox Palo Alto Research Center (PARC) in about 1974. Dick called his file save and restore routines *savpa* and *respa*.

nual pages using the "alpha" and "RGBA" terminology, ready for use. I had called the new channel "alpha" because of the classic linear interpolation formula, $\alpha I + (1 - \alpha)J$, that controls the amount of interpolation between two entities—in this case, two images I and J. All Ed had to do was write the alpha coefficient into a fourth framebuffer (we had six 8-bit framebuffers at New York Tech at the time). Then I would save the four framebuffers—for Red, Green, Blue, and a new one for Alpha[7]—into a file with the new code, called *savpa4*.[8] Then Ed or I *or anybody* could use the newly revised restore routine (called *getpa*) to composite the file image over an arbitrary image already in the framebuffers. *getpa* would detect that the enclosed image had a fourth channel and use it to do compositing, as the image was read from the file. That was it. The integral alpha channel had been born.

The "or anybody could use" phrase above is key. The integral alpha channel severed the image synthesis step from the compositing step, and this changed how digital compositing was done forever. When we started Lucasfilm graphics a couple of years later, in 1980, it was RGBA from the outset. The original framebuffers there were RGBA and all software was written to honor RGBA. This was the stage upon which the second act of innovation was played.

Invention of Premultiplied Alpha

Although we had added the alpha channel to our thoughts and computations and hardware at Lucasfilm in the early 1980s, we still did not fully understand it. In particular, it wasn't until my Lucasfilm (later, Pixar) colleagues Tom Porter and Tom Duff invented the *matting algebra* (Porter & Duff 1984), introducing the new concept of *premultiplied* alpha, that the

[7] We called three 8-bit framebuffers ganged together an RGB framebuffer and four an RGBA framebuffer.

[8] The earliest dated documentation I have for this code is dated 13 Jan. 1978. Ed was preparing for SIGGRAPH 78. SIGGRAPH typically has a paper due date of early January of the corresponding year, so this is probably about when the invention actually occurred although it might have happened in Dec. 1977, to avoid the last minute crunch against the paper deadline. I was also preparing a paper for SIGGRAPH 78. The date on the submission is 6 Jan. 1978, and the code I used to generate figures for the paper is dated 28 Dec. 1977.

power of the idea started coming into focus. Since their solution is an excellent example of how carefully distinguishing geometrical ideas from sampling ones bears imaging fruit, I shall describe the Porter-Duff discovery in detail.[9]

There are two ways to think of the alpha of a pixel. As is usual in computer graphics, one interpretation comes from the geometry half of the world and the other from the sampling half. Geometers think of "pixels" as geometrical areas intersected by geometrical objects.[10] For them, alpha is the percentage *coverage* of a pixel by a geometrical object. Imagers think of pixels as point samples of a continuum. For them, alpha is the *opacity* at each sample. In the end, it is the imaging model that dominates, because a geometric picture must be reduced to point samples to display—it must be *rendered*. Thus, during rendering, coverage is always converted to opacity, and all geometry is lost. The Porter-Duff matting algebra is based on a model that is easiest to understand by alternating between the two conceptions.[11]

The elementary imaging operation that we wish to elaborate is called, in (Porter & Duff 1984), the **over** operator. It captures the notion of compositing image J over image I, where either I or J or both may be partially opaque. For ease, we will think of images I and J as being rectilinear, the same size, in register, and each having four channels, three for RGB color and one for alpha—that is, opacity. Images were always still rectilinear at that time.

[9] Ed Catmull, Tom Duff, Thomas Porter, and I received an Academy Award for our "pioneering inventions in digital image compositing" on 2 Mar. 1996, for the work described here. The specific award is the Scientific and Engineering Award of the Academy of Motion Picture Arts & Sciences.

[10] A little square is a very common model for the "pixel" among geometers. I place this term in quotes to remind us that this is not a pixel (a sample) but a model for possible geometric contributions to the final sample. The last thing I want to promulgate is the notion that a pixel is a little square. See appendix A.

[11] The Porter-Duff paper is an excellent example of why the little square model for contributions to a pixel has become confused, in the geometry-based computer graphics world, with the pixel itself. All illustrations in that paper use the little square model. A unit circle could have been used equally effectively, however—or any other unit area region.

ORIGINS

Think of the following geometrical model: A "pixel" is an area α percent covered by an opaque geometrical object with color A.[12] Thus the amount of color contributed by that area is αA. That is, we average the color over the "pixel" and come up with a single new color representing the entire area—the color αA is a point sample.

Now think of another opaque geometrical object with color B added to the original "pixel" area. Disregard for a moment the other geometrical object there. Assume that the new geometrical object has coverage of the "pixel" equal to β. So the "pixel" is contributing color βB due to this object. This again is a point sample representing the color of the second object.

But now we use the geometry model to conceptually combine the contributions of the two objects in the "pixel" area. The second object is allowing only $(1-\beta)$ percent of the "pixel" area to be transparent to any objects behind it. We simply ignore the actual geometry of the two objects at this point and assume that, in general, the "pixel" is allowing $(1-\beta)$ times the color from behind, αA, to show. This is added to the color due to the top object βB. So the total color of object with color B over object with color A is $\beta B + (1-\beta)\alpha A$.

Notice that this result could be completely wrong if the geometry of the second object exactly coincided with that of the first. The bottom color would not contribute at all to the final color in this special case. So the model we are using is an approximation for the general case of combining two images where we no longer have any idea of how the alpha at a point was determined. In an image there is no way to tell whether a point sample with a partial opacity comes from a partially transparent surface or from an opaque surface partially occluding the area represented by the point sample.

The formula just derived from basic principles is this: For composite color C obtained by placing a pixel (no quotation marks needed) with color B and alpha β over a pixel with color A and alpha α:

$$C = \beta B + (1-\beta)\alpha A = \beta B + \alpha A - \beta\alpha A.$$

[12] A color typically has several *components*, here usually RGB. Each operation described here would therefore have to be performed on each component.

Notice how many multiplies this formula implies—three[13] at each pixel for each color component. Considering that this formula is basic to computer graphics, and that multiplies are expensive,[14] Porter and Duff observed that it could be reduced to one multiply per pixel per component if the alphas were *premultiplied* times the color of an image. That is, if the color channels of image I contained, not color A, but weighted color αA, and similarly for image J, then the formula above reduces to

$$C' = B' + (1 - \beta)A' = B' + A' - \beta A'$$

where the primes indicate colors have been premultiplied by their corresponding alphas. Such images are said to have *premultiplied alpha*.[15] Of course, it is the color channels that are different, not the alpha channels, despite this terminology. Next I will show that premultiplied alpha was more profound than any of us knew at the time.

Invention of the Sprite

Although we had added premultiplied alpha to our thoughts and computations and hardware at Lucasfilm in the early 1980s and Pixar in the 1980s and early 1990s, we still did not fully understand it. The evolution of the concept occurred in distinct stages.

In the earliest days of computer graphics alpha was not even called alpha, but rather was called a "matte" or "mask." It captured the old filmmaking idea of a matte as a separate piece of film used to combine two other pieces of film. Typically a matte was not in the same storage file as the color image or images to which it was to be applied. It was not integral.

In the non-premultiplied alpha case—the original conception of the integral alpha—a rectilinear color image had a rectilinear alpha channel that defined the opacity of each pixel. This was simply a convenience.

[13] Two, actually, with a little rearrangement and a temporary variable: $T = \alpha A, C = \beta(B - T) + T$.

[14] They were especially expensive in the 1980s. Now we would just like to avoid extra steps.

[15] Porter and Duff actually called this case *associated alpha*, and the alternative *unassociated*, but I always had trouble remembering which was which, hence began using "premultiplied" in place of "associated."

ORIGINS

In the premultiplied alpha case—the modern conception of integral alpha—a rectilinear color image had a rectilinear premultiplied alpha channel that defined the opacity of each pixel. This simply sped up compositing computations. Appendix B argues the case for premultiplied alpha, emphasizing its elegance.

The mental baggage of the rectangle dominates in all three of these formulations, reinforced by the common use of a (rectilinear) array for storage. The breakthrough idea—that of the shaped image, or *sprite*—came from this simple observation: *Premultiplication by alpha completely clears transparent pixels of any current or future information*. A pixel with an alpha of 0 requires, in the premultiplied alpha case, that all of its color channels be 0 too. The transparent pixels cease to exist conceptually since they are forever unusable. And once one ceases to think of empty pixels, it is an easy leap to shaped images. It is the same act as ignoring the empty space around the lines and vertices of a triangle in the geometrical world when one thinks of a triangle.

You might argue that a sprite is stored in a rectilinear piece of memory, so the rectangle has not really "ceased to exist." But this rectilinear memory is just a convenience and not required. I call a pixel with premultiplied alpha of 0 a *clear* pixel. Notice that clear pixels do not have to be stored. Usually they are, in applications thus far anyway, but there is no reason other than convenience to do so. There is no useful information in a clear pixel. It doesn't exist unless a programmer chooses to force its existence for storage or programming convenience.

It wasn't until I began writing a language for image computing that the profundity of the idea began to hit me: The premultiplied alpha concept had rid us of the rectangle! This was really a matter of identifying what was already in plain sight—just naming it—rather than an "invention." Since many ideas in sprite theory evolved from the exercise of writing a language of image computing, it is worth describing that attempt.

SPRITE THEORY

ICEMAN—A First Attempt at a Theory

In the beginning was the understanding that a theory was needed at all. It came from my realization in the 1980s that the "image processing" market had not taken off because it had no center—the sampling-based half of the 2D computer picturing world was undefined. The 2D geometry[16] market—also known as desktop publishing—was founded on the careful definition of 2D geometry embodied in the Adobe POSTSCRIPT language, but there was no corresponding accepted definition of 2D image computing. Every company or institution in the business had an internal idea, often vague, of a model that, of course, differed from the others—if not from other models within each company or institution itself (as I can attest). So I spent about a year at Pixar defining a language, called ICEMAN, to accomplish for images what POSTSCRIPT did for 2D geometry.[17] But I have a story to tell about the original insight that puts the problem in perspective.

In 1988 I gave the "Midnight Sun Lecture" at a conference in Trømsö, Norway, about 200 miles north of the Arctic Circle, June 21, Summer Solstice.[18] Besides being the most fun I ever had at a conference, I obtained advice there that changed my life and made my fortune, and from which this lecture proceeds. Pixar was only two years old and was selling hardware to keep everyone employed while we waited for the price of computation to drop enough to make animated films economically feasible. We were having trouble in the marketplace distinguishing our product, the Pixar Image Computer, from the product of Silicon Graphics based on Jim Clark's "geometry engine."[19] Many thought we were competitors although we knew we were not. In my talk I made a big point about how the two products differed, going on at some length. What I haven't told you is that both I and my audience

[16] I will everywhere use 2D to mean "2-dimensional," 3D for "3-dimensional," and generally nD for "n-dimensional," for arbitrary non-negative integer n. I occasionally use the form $(n+1)$D for n spatial dimensions and one temporal dimension.

[17] The ICE in ICEMAN stands for Image Computing Environment. The MAN reflects a similar usage in another Pixar product, RENDERMAN. I called it temporarily VAIL, for Volume And Imaging Language.

[18] Second International Conference on Vector and Parallel Computing.

[19] Jim Clark had been our colleague at NYIT, and he taught me 3D computer graphics.

were highly fueled for this event by liberal amounts of aquavit. It was clear that I was to entertain, firstly, and inform, secondly. I was speaking at midnight, after all. I recall it as one of the greatest speeches of my life. That memory is suspect, but what happened at the end of the talk is not. A man came forward, obviously under the influence (of the aquavit, not me), and managed to say the fateful words, "I think all you have to say is that you don't do geometry." I knew instantly—even slightly stupified on aquavit—that he was right, that he had happened onto the correct marketing message, always a difficult accomplishment in business. Because, you see, we had been emphasizing that our Pixar image "supercomputer" could also do geometry. That was a mixed message that put us seemingly in competition with Silicon Graphics.

The very next day, after the intoxication had worn off, I sat down in my hotel room[20] and began writing a theory of image computing, defining what it was that we actually did. I thought of it as defining the POSTSCRIPT of imaging. See (Smith 1988a).

I was also just mastering C++ and object-oriented programming. So I took as my "masters class" in OOP the definition of a language for image computing and an implementation of it in C++. That language was ICEMAN. The definition of the language forced me to understand image theory and draw distinctions that had never been carefully drawn before, even among my colleagues. I then wrote a prototype application based on the concepts of ICEMAN and called it Composer. From this I started Altamira Software Corporation, as a spinout from Piar, which productized Composer as Altamira COMPOSER, and then sold it (and the company) to Microsoft in 1994.[21] To explain the sprite theory, I will often take examples from ICEMAN and COMPOSER.

The ICEMAN language remained with Pixar when I left to form Altamira Software. Altamira COMPOSER was not based on the language ICEMAN, other than that it shared some definitions with it, and shared of course the inherent model too. I make this point to justify why I am call-

[20] With hammers and sickles displayed out the window, to my amazement, on the smokestacks of ships in the harbor—the Berlin Wall would not fall for over a year.
[21] Microsoft's imaging products, IMAGE COMPOSER, PHOTODRAW, and others came from this base.

ing ICEMAN a first attempt at a theory. Although it was a pretty thought—the language of imaging—it turned out to be overkill, as I will explain.

I captured shaped images in ICEMAN with the **image** type. In Altamira COMPOSER, I captured it in the *image object*, and called them that in Altamira marketing. This got confused with "layers" in the market, a restricted and rigid concept introduced by Adobe as in response to my image objects.[22] Layers can be simulated with image objects but not vice versa. So here I am going to call the shaped image, or image object, a *sprite*. I wish I had thought of marketing them as sprites at Altamira. This would have simplified the marketing message and perhaps kept the confusion with layers at bay.

So it was alpha intimately in the pixel, the integral alpha channel, and premultiplied alpha—with transparent pixels forever superfluous and hence conceptually nonexistent—that led eventually to the shaped image, or sprite. The result has been to free the image from the tyranny of the rectangle and to make it—or more properly, a sprite—coequal to a geometric object. This is one example of digital convergence between two quite different objects, once they are correctly represented digitally.

As will be shown, the creation of the sprite (in the generalized meaning here as a shaped image object) has changed the notion of image computing from "image processing"—which connotes doing things to a static monolithic rectilinear image—to "image composition"—which connotes a space full of floating, shaped, partially transparent sprites that can be rearranged in spatial position and depth arbitrarily. Of course, the ability to do arbitrary image processing or editing on each sprite is still completely available but is no longer necessarily the focus of an inte-

[22] To get a feel for the limitations of the layer concept, try to generalize it to 3D. Freely floating 3D sprites easily generalize, but layers don't. Furthermore, in 2D, layers usually connote a stack of rectilinear images, all of the same size, in register. One has to carry around the mental baggage of the transparent portions as being part of the layer. One falls off the edges of layers—i.e., gets cropped to the rectilinear background. There is no conceptual problem with having two sprites at the same depth but in layer terminology, one has to assign the two sprites to the pre-existing layer to model the concept. This is unnecessary machinery.

raction with images. Most of the operations have to be rethought, however, to handle alpha and shape information.

Origin of the Word "Sprite"

The sprite terminology was first applied to very simple icons such as the gobbling head in the early PAC-MAN game. I had noticed already that the sprite was the nearest thing at Microsoft to what we were using at Altamira, but failed to generalize their term and apply it to our image objects, thinking that sprites were still—as they were originally—dumb, extremely simple, rigid icons on personal computer screens, with jagged edges and few colors, exactly what we did not want associated with our image objects. Interestingly, however, it was an interview in *Scientific American* with Nathan Myhrvold where he mentioned briefly a conversation with Bill Gates about a generalized notion of sprite that alerted me that I should visit Nathan and tell him about our concepts before he "did it wrong." This visit resulted in the purchase of Altamira by Microsoft, and belatedly in my adopting the extended notion of sprite as the name for shaped images.

In the next part I begin to lay out the fundamental tenets and defitions of the sprite theory of imaging. I will more carefully lay out distinctions hinted at already. In particular, I will distinguish sampling-based picture making from geometry-based and carefully explain how to convert from one to the other, and I will differentiate creative space from display space. This will supply enough machinery to enable defining the fundamental notions, those of image, for rectilinear sampled pictures, and sprite, for shaped sampled pictures. Succeeding parts will describe "algebras" for working with these entities.

FUNDAMENTALS

Sampling vs Geometry

An important contribution of the sprite theory is a careful delineation between imaging, or sampling, and geometry. I have drawn this fundamental distinction many times—for example, (Smith 1988a), but it bears repeating because the confusion is still common. See Figure 1.

I call the act of making pictures with computers *computer picturing*. There are two different ways to make pictures with computers—that is, two different modes of computer picturing. The one most popularly understood is that based on geometry, often called "computer graphics" or "CGI" or "3D synthesis." This includes, for example, *Jurassic Park*, *Toy Story*, or *Ratatouille* graphics from Pixar and Disney. The other is based on sampling theory, often called "image processing," "image computing," "image editing," or just "imaging." The digital pictures sent back from the Voyager and Cassini planetary flybys or from the Hubble telescope are of this type, as are digitized photographs from your digital camera, and digital videos as on <YouTube.com>. The Visible Human from the National Library of Medicine is an excellent example of 3D or volumetric imaging. In the sampling-based way, there is no geometry involved at all. In the geometry-based way, there is no sampling until the rendering step required for the display of the result.

The sampling vs geometry distinction is fundamental. The two paths, *geometry-based* and *sampling-based* (or *sample-based*, or, equivalently, *image-based*, or *imaging-based*, for the latter), have different mathematical bases (geometry and sampling theory), different heroes (Descartes and Nyquist, say), different histories, different journals and conferences, and so forth. Ivan Sutherland is often given credit for fathering computer graphics, but even if a true fact (and I don't believe it is),[23] then it only applies to the geometry half of computer picturing.

[23] Perhaps the most accurate statement is that Sutherland was the first with interactive *editing* of geometrical computer graphics. He was certainly preceded by others with interactivity and geometry. And the development of sampling-based computer picturing proceeded in parallel with completely different but simultaneous players.

FUNDAMENTALS

The reason for confusion is easy. We cannot see geometry. Geometry is abstract. In order to see it, we have to convert it into an array of samples, called an image (and the samples are called pixels, of course). This conversion step is called *rendering*. Since both approaches to picture making with computers result in an array of pixels, many—especially non-technical people—cannot distinguish the two processes. I will argue that creation is distinct from display, and it is in the creation step that the two modes of picturing are definitely and clearly different.

The distinction being drawn is really between the continuous and the discrete. Geometrical descriptions are continuous and use the real numbers. Sampling descriptions are discrete and use the integers, especially in the case of pictures. Geometric descriptions, when they suffice, can be extremely succinct. Sampling descriptions, can describe many more things than geometry, but suffer from a definite lack of succinctness. The point is that both are equally valid, but different. I shall be very careful to distinguish geometrical concepts from imaging concepts below. You might think me excessive in this, but I almost daily see and hear confusions that are directly due to a lack of care at this boundary. Once we are comfortable with the distinctions, then it is straightforward to implement the digital convergence of geometry and sampling.

Some words or notions that put me on alert that the geometry–imaging confusion is lurking are these (each is followed by a correction or criticism):

A pixel is a little geometric square.[24]

No. A pixel is a point sample. If it has any geometry at all, it is a point. See appendix A for my detailed argument against the little square misconcept of a pixel.

[24] The little square has been extremely important to computer graphics—we wouldn't be where we are today in 3D synthesis without it. It is a simplifying model that represents contributions to a pixel. The mistake is to *identify* this simplifying model with the pixel itself, or equivalently of identifying the geometrically-defined area of a model sampled with the sample that represents that area.

A pixel is located on the half pixels.

No. Samples are array elements with indices as "location." There is no such thing as a half pixel. The notion that there is one comes from the pesky little square model. See appendix A.

A display has non-square pixels.

No. Again, pixels are point samples, so "non-square" makes no sense. The correct idea is captured by the *pixel spacing ratio*, the sampling distance in the horizontal dimension divided by that in the vertical. So, instead of saying nonsensically, "non-square pixel spacing," say instead the meaningful, "the pixel spacing ratio is not 1." See appendix A.

Images have regions of interest.

No. What this usually means is a geometrically defined region of an image, where the mapping of the geometry to the sampled image is assumed obvious. It never is.[25] The alpha channel always captures the notion accurately and with much more generality. The alpha channel can always be used to make a geometrically "defined" region precisely defined.

An image is a rectangle or rectangular picture.

No. An image is, ignoring the alpha channel now, a rectilinear array of samples. It does not reconstruct, by the Sampling Theorem, into a rectangle or rectangular picture,[26] in any but the simplest use of that theorem—with the worst reconstruction filter, a box.[27]

[25] If you don't believe me, try this exercise: Consider a rectilinear image that is 3 pixels high by 3 pixels wide. Assume that the "region of interest" is the entire image. Define the rectangle that defines this region of interest. I can immediately think of several possibilities. I bet you can too. Which one of these is the "obvious" mapping? Does your definition assume little square pixels with centers? Does your rectangle use the notion of "half-pixels"? What is the geometric width of your rectangle? For example, is it 3 or 4? Does your definition take sophisticated reconstruction filters into account? Or just the too-simple box filter? And that's just for a simple rectangular region of interest.

[26] See the section below entitled Continuous Operators on Discrete Sprites (page 29) and its Figure 2.

[27] Well, the worst is really no filter at all: point sampling.

FUNDAMENTALS

Creative Space vs Display Space

Another fundamental distinction that the sprite theory makes is that between creative space and display space. I believe that this distinction is one of the most profound contributions of the computer to the arts. It is an obvious distinction in 3D computer graphics synthesis. There one models in 3D unbounded space with abstract geometrical objects, then displays views of this *creative space* that are 2D and restricted to the size allowed by some output medium such as 70mm film or 1080p high-definition video. The choice of *display space* is a separate act from the creating or modeling in creative space. It is a separate creative step, in fact. Another way to put it: Many display spaces can correspond to one creative space.

We shall borrow this distinction completely into the imaging domain. This is new. Nearly all imaging applications (for example, PHOTOSHOP) confuse the two. For example, most popular imaging applications equate display space to the rectilinear image being edited. One opens an image (meaning rectilinear with no partial transparencies) and this maps to its own window on a display screen. Open another image and it is mapped to its own window. The fact that they could be two sprites in the same creative space is thus outlawed from the beginning.

This is to be contrasted to a 2D drawing program for example. Here, just as in the 3D synthesis case, one creates in a space which is unbounded 2D (or unbounded 3D). Different geometric objects (a square, a triangle, etc) can be placed in that space and moved around relative one another, placed in different depth order, grouped, aligned against one another, and so forth. In the sprite model, image sprites can be dealt with exactly analogously. And once this is absorbed, then it is obvious how to implement the digital convergence of 2D image objects (sprites) and 2D geometrical objects. Simply put the two different objects in the same creative space and render them appropriately to display space, that is, with the renderer suitable to the type of the object.

The Sampling Theorem

I refer to sampling so often here that I want to make it very clear what exactly I mean. Geometry, on the other hand, I will take as a given. We've all

learned geometry, if not formally, since we were children, but sampling is more subtle and not part of a general education. It is also pervasive in the modern world and deserves higher stature as an educational fundamental.

Sampling is the act of choosing a discrete set of examples from a continuous set and representing the continuous set with the discrete set. That is, a sampled representation "throws away" an infinity of points, preserving only a more succinct and useful set. How is this possible? The Sampling Theorem tells us how. The fact that it is true is what makes the digital world—particularly digital images and digital audio—possible. Without sampling the digital worlds of television, movies, and audio would cease to exist.

I do not intend this lecture to be about the mathematics of sampling theory, so I will present the Sampling Theorem intuitively. You will have to trust me that this loose way of speaking is thoroughly backed with hard-core mathematics, and that I have preserved enough of the caveats to make the loose understanding of it I hope you get here adequate. The mathematicians among you will cringe, but I think it is important that this subject be widely understood and it will not be if all the mathematical niceties are met.

The first point I wish were more widely understood is this: the world is composed of frequencies. It is a strange point to most non-engineers and non-scientists, but it figures in any appropriate use of the Sampling Theorem. It is also fun in a way I will show you and more intuitive than first glance might suggest.

In looking around yourself you might think that what is being presented to your eyes is a continuous field of colors—at every point in your view there is a color. This infinity of colored points constitutes what you can see at any moment. Now all engineers are taught early in their careers that a continuum, such as your visual field, can alternatively be represented as a sum of waves of different frequencies. This is the alternative view I want you to be comfortable with.

I'm looking around my desk as I type. I see the Venetian blinds. They are equally spaced. That is, they form a spatial wave of a given frequency: The slats appear at the frequency of about one slat every inch and a half

or so. Then I spot the books on my bookshelf. Their spatial frequency is about one book per inch or so. They are not equally spaced but they appear in a frequency range of, say, one book every half inch (monographs), down to one book every five inches (dictionaries). Then I look at the keyboard on which I am typing. The keys have a spatial frequency peculiar to keyboard keys. The floorboards have their spatial frequency. The room corners have their (much lower) spatial frequency. I look between the slats of the Venetian blinds at the trees across the street. Their leaves have a high spatial frequency or frequencies.

Perhaps that is enough intuition building to understand the wonderful leap that was made in the 19th century:[28] There is a sum of (waves of different) frequencies that is exactly equivalent to the visual field as a collection of colored points. In general, there is a frequency equivalent to the spatial color field. A full understanding requires a statement of the theorem giving the equivalency, but that would take us too far afield. The takeaway I want you to have is that scientists and engineers often go back and forth between a frequency representation of the visual world and a spatial color intensity representation. Actually between any (nice) continuous field and a frequency representation of it. (That "nice" is included for the mathematicians—it is possible to find abnormal fields which fail, but the ones we shall be interested in do not, and are therefore "nice.") I will nearly always use for examples a continuous visual field, perhaps changing in time, as the continuous field being sampled, but occasionally I will use an audio field. The summary idea is this: It is often easier to think in terms of frequencies than color fields (or sound pressure fields), and, importantly, the two ways are equivalent.

Here's a common way we talk about frequencies in a picture: If there are sharp edges in a picture, then we say that the picture contains high frequencies. What this means is that one would have to add in waves of very high frequencies, in the sum of waves model, to have a sum which

[28] It was Fourier who made this amazing discovery in the early 19th century. He was trying to solve the heat distribution problem in cannons at the time, to keep them from blowing up when fired. Fourier analysis is fundamental to all modern engineering since it is often to easier to work in the frequency domain than the spatial or temporal domain.

resulted in a sharp edge, in the color field model. The takeaway intuition here is: sharp edges imply high frequencies.

Here's another way to think about the frequency/color field equivalency: The JPEG (.jpg) type of digital image file format is able to greatly reduce image file size by first converting the color field to its frequency equivalent, throwing out those frequencies that are not used or used only slightly, then converting back to the color field equivalent to the reduced frequency representation. This works spectacularly well. In other words, every time you save one of your photographs as a JPEG file, you yourself have invoked the frequency equivalent of a color field. So this should give you enough intuition about the omnipresence of frequencies in the world about you to let me explain the all-important Sampling Theorem.

The Sampling Theorem makes this lecture possible. I shall express it intuitively, as I did the frequency domain vs. spatial domain equivalency:

> A continuous field can be represented perfectly by a discrete set of uniformly spaced samples, if (1) there are no frequencies in the field higher than half the sampling frequency,[29] and (2) the reconstruction (of the continuous field from the discrete samples) is carried out with a filter of the "right" shape.

Disregard for the moment constraint (2). To understand it I would have to explain "reconstruction" with a "filter." I would also have to explain what the "right" shape of the reconstruction filter is. It is often the case in the real world, and imaging in particular, that we don't actually have to do reconstruction explicitly, if we have obtained discrete samples according to constraint (1) only. Our display devices do the reconstruction, although not perfectly. But correct reconstruction is fundamental to image computing, so I will return to it.

For now, just consider the Sampling Theorem so far as obtaining the samples is concerned, under constraint only that the sampling frequency

[29] The "half" might be confusing. To get an intuition for it, think of a wave as consisting of alternating up parts and down parts. It makes sense, intuitively at least, that it takes two samples, one in the up part and one in the down, to capture the information converyed by that wave. Hence sampling has to occur at twice the highest frequency, equivalent to the statement above.

be at least twice the highest frequency in the continuous field. This is the only place where I actually ask you to understand that a visual, spatial field is equivalent to a sum of waves of different frequencies. Roughly, the Sampling Theorem states that that in order to accurately represent a scene with tree leaves in it, you have to sample the scene at least once per leaf—the more dense the detail, the higher the frequencies needed.

Usually you have a fixed sampling rate and can only get a representation as accurate as that fixed rate allows, by the Sampling Theorem. If you have a 10 megapixel camera, then you cannot capture a scene with 15 million changes in it, but a 20 megapixel camera can. This may seem intuitively obvious, but it is really quite profound. Remember that the pixels in your camera are not little squares. They are point samples, and the scene being reproduced is continuous whereas the point samples are spatially separated and discrete. This is the magic of the Sampling Theorem.

At this point you might say, "But the displayed pixels are are not points. They spread out and touch one another on the little display screen on the back of my digital camera." This is where reconstruction comes in. Although you do not explicitly reconstruct the image you have made, the display device attached to your camera does, or the computer monitor on which you view your digital photos does. A characteristic of a real-world display device is that it physically cannot display a point. It causes a point to spread. This is equivalent to reconstructing the sample point with a reconstruction filter, although not the ideal one.

The ideal reconstruction filter called for by the Sampling Theorem is of the form $(\sin x)/x$, where x is related to the sampling frequency. This classic filter, called the $\mathrm{sinc}(x)$ filter, will not be pursued here because it is an ideal filter, infinite in extent, and hence not useful for computations using a finite computer. When we do explicit reconstructions, we shall use finite approximations to the ideal filter, and we shall be very careful about them. Suffice it to say here that the magic of the Sampling Theorem comes about by substituting, in a devilishly clever way, the infinity between sample points with the infinity of the reconstruction filter. I shall not say more here about how the Sampling Theorem works,

but I would like to try one more time to make you comfortable with the frequency version of the universe.

Consider audio instead of video. It is well-known that music, in fact all audio, consists of frequencies of sound. Roughly speaking, each note is a frequency. The result of adding up all those simultaneous notes (frequencies) and presenting them to our ear is a sound field, a continuous audio signal, that we interpret as, say, music. Digital audio works by sampling the 1D temporal sound field exactly the same way that a 2D spatial color field is sampled, by appropriate use of the Sampling Theorem. The two worlds are so similar that the "group now known as Pixar" included digital audio scientists among its earliest incarnation at NYIT on Long Island, and also in its second incarnation at Lucasfilm in Marin County, California. We all understood we were involved in the same technology, but applied to two different continuous fields. We were invoking the same magic.

Definition of Image

An "image" will be defined to capture the lay notion of a rectilinear picture, then "sprite" will be defined from "image" to capture shape.

> *An* image *is a finite* n*D rectilinear array of pixels of identical type. A* pixel *consists of one or more channelvalues of identical type, where a* channelvalue *is a number representing a sample.*

This deceptively simple definition hides a host of implications concerning finiteness, dimension, shape, type, sampling rate, alignment, and uniformity. This discussion should also make it clear why no two organizations completely agree in general on the definition of an image, and hence the need for a model.

An image must be finite. Strictly speaking, it must be representable in a digital computer. This means that an image is finite in extent in each of its dimensions and in the number of bits per channel. The definition is still quite broad. In an actual implementation of the sprite model, only certain available data types—for example, integers, floats, bytes—are available for representing channels, hence images.

FUNDAMENTALS

An image is nD, with $n \geq 0$. The dimensions are not necessarily spatial. Our model principally provides support for 2D images, time sequences of 2D images (movies, books), 3D images (volumes), and time sequences of 3D images (volume movies). The model supports 1D images mainly to the extent they are subimages (scanlines) of 2D or 3D images, and it supports 0D images to the extent they are trivial subimages (pixels) of higher dimensional images. But time and space are just names for the dimensions; the model does not care what they really represent (although we will make a sampling assumption as will be seen). And subimages are images, so the model supports 0D through 4D images and is theoretically capable of higher.

An image is rectilinear. This means that its support is rectilinear, where the *support* of an image is the set of locations at which its samples are taken. The support for a 1D image is a finite set of points along a line segment. The support for a 2D image is a finite set of points on a rectangle-bounded plane. The support for a 3D image is a finite set of points in a volume bounded by a right rectilinear parallelepiped. And so forth. Clearly, non-rectilinear images are necessary—for example, as brushes in a paint program. The sprite theory handles arbitrary shape with an imaging notion, not a geometrical one, and will be discussed subsequently.

An image has pixels of identical type, and its pixels have channelvalues of identical type. The number of channelvalues per pixel is called its *ply*. The type of a pixel is determined by the number and type of its channelvalues. For example, a pixel in a prepress application might have five channels—representing yellow, magenta, cyan, key, alpha (YMCKA)—where the channelvalues are 8-bit unsigned bytes. All pixels in a single image must have this same type. In our prepress example, a pixel with ply more than, or less than, five is not allowed in an image of these pixels. An image may also be thought of as a list of channels, where the ith channel of the image is an array of all the ith channelvalues of the pixels comprising the image. An image has the same ply as its pixels of course. The model provides utilities for combining channels into a thicker (in the sense of more channels, or higher ply) image, or for extracting chan-

nels from an image to form another thinner image. With these functions, arbitrary combinations of images with different pixel types can be effected so long as they have channelvalue types in common.

It is easy to imagine channels with different types. They are represented in sprite theory as separate images bound together with a higher-order data structure.[30]

An image consists of samples. A fundamental assumption of the model—captured by the final phrase in the definition: *representing a sample*—is that each channel of an image can be reconstructed, using the Sampling Theorem, to retrieve the continuum which it represents—for example, a color separation, an electromagnetic field, an airflow over a wing, or a height or depth field. This does not mean that such a reconstruction will ever happen, nor does it imply that the samples were taken correctly in that the continuum was low-pass filtered for removal of inappropriate frequencies. Neither does it imply that anything fancier than point sampling will actually be employed in an image computation. The assumption of sampling does restrict the class of things which can be called images. For example, a 4×4 matrix is generally not an image. A program—a 1D list of numbers—is not an image. A list of telephone numbers or polygons is not an image. And since samples must be numbers, the following are also not images: a tiled floor, a chessboard, a deck of cards, a Rubik's cube. More pertinently, an image of geometrical objects is not an image unless it is a digital representation of those objects—that is, a sampling, or "rendering" or "scan conversion," of the continuous picture of the objects. Similarly, in the the model sense, a photograph or painting is not an image unless it is digitally represented. A goal and important contribution of the model is to properly take care of filtering while hiding the difficult details of this task from the user of an application, unless the user specifically wants to deal with the subject. It should be added that there is no way for the model to "know" whether an array of numbers is an image or not. Many of the capabilities of the

[30] Altamira COMPOSER routines always had a first argument pointing to an "imagestruct" which was used for such higher order bindings, and for threading all "global" variables to the routines.

model are undoubtedly useful for computing on these non-image arrays as well.

An image has all channels at the same resolution. The sizes or extents of the dimensions can each be different, but whatever the *n* sizes are in one channel must be the same in all other channels of an image. For example, a 2D image with red, green, and blue (RGB) channels which has a red channel of size 1280×768 samples, must have the green and blue channels also at sizes 1280×768. There are common cases that seem to violate this part of the definition. For instance, a depth channel associated with our RGB example might be "supersampled" to have 8x8 depth samples per each color sample. Or in a graphic arts example, the line art or text might be a single-bit channel at a resolution six times as high as the "contone" (continuous tone) color channels which might be CMYK channels of 8 bits each, or 16 bits each. Our model requires that images at different resolutions be treated as separate images.[31] An application binds them into a single entity as a higher-order structure.

An image has all samples of a pixel aligned at a single location. This means in our example that the RGB samples in a single pixel correspond to the color at a single point of the picture represented by the image. Strictly speaking, the definition has no requirement that this be so, but the model assumes it. An application built on the model might elect to ignore this assumption. As for different resolutions, the model handles nonaligned channels as different images bound together, by an application program, with a higher-order data structure. In graphic arts, for example, the four color separations of an image are often converted to halftone spot arrays which are rotated relative one another. If these halftone separations are represented as numbers aligned with the rotated axes, then they cannot be channels of a single image; a higher-order image structure is required. But if the rotated halftones are actually represented with very high resolution, aligned bit arrays, then they may of course be treated as channels of a single image.

[31] Or that a multi-resolution representation be hidden from users using the object-oriented paradigm.

An image has all samples taken uniformly in each dimension. The rate or spacing can differ between dimensions, however. Again, strictly speaking, the definition does not require this, but the model assumes it. To be clear, the model does not assume resampling cannot occur nonuniformly, as in image warping, but it does assume that the input and output of such a computation are uniformly sampled. An application can elect to ignore the assumption. An example where this might be appropriate occurs in medical volume imaging. CT slices are frequently acquired at nonuniform spacings through a patient's body. An application program in this case would have to deal with input volumes of nonuniform sample spacings.

Finally, the model of image assumes samples are taken uniformly and aligned with its rectilinear edges—that is, on the nodes of a rectilinear grid. So hexagonal sampling grids, for example, are not used by the model. Again, an application could elect to disregard this assumption.

Despite all the subtleties in the definition, the model definition on the whole captures most of the intuitive notions while maintaining a remarkable simplicity. This is crucial for widespread applicability.

The actual data representation of an image is not specified, in the spirit of object-oriented programming. It could be an array, a set of arrays, a tiled (paged) set of arrays, a multi-resolution and tiled set of arrays, etc.

Despite the remaining generality of the definition of image, in practice—and in the remainder of this lecture—an image will always be taken to consist of color samples as pixels with the addition possibly of an alpha channel to each pixel. This is essentially an application of the restriction that pixels represent samples of a continuum, where the continuum is a color field. In particular, I shall nearly always use an RGB image or an RGBA image as exemplar—one with Red, Green, and Blue channels for color, and possibly an Alpha channel for opacity. I assume that the theory is easily applied to images of different structure or meaning. It will become clear I believe that much of the machinery of the theory is independent of the pixel structure and hence widely applicable.

Here then is the definition of the "image" actually used in practice in this lecture and the one actually realized in Altamira COMPOSER:

An RGBA image *is a finite 2D rectilinear array of pixels of identical type. A* pixel *consists of* 4 *channelvalues of identical type, where the first three represent a sample of an RGB color field, and the fourth A represents a sample of the corresponding opacity, or alpha, field. An RGB image is an RGBA image without the alpha channel.*

So here it what has happened in this part: I started with a simple, clean definition of image. I suspected that many would find fault with this definition, and I assumed that the fault would generally be found to be a lack of generality. So I discussed at length each decision that contributed to the form of the definition adopted. Finally, I tackled the reverse problem, that the definition was still too general. Rather than make a clumsy specific definition,[32] however, I have opted to stay with the original generality (restricted as it is) and simply state the actual further restrictions. I want it to be clear that the theory applies more generally than to the restricted images I shall actually use here for explanation.

We are now ready for the definition of the key entity in the theory, the sprite, or shaped image.

Definition of Sprite

As a brief review: Non-rectilinear shape of an image was traditionally determined by another image, called a *matte*. The matte can be a separate image or reside in a channel of the image to be shaped. Such a channel is sometimes referred to as a *matte channel*, or equivalently an *alpha channel*. In sprite theory the matte is always an (integral) alpha channel, and it is premultiplied: Where it is 0, the corresponding image does not exist.[33] Where it is 1,[34] the image is opaque. Where fractional between 0 and 1, the corresponding image is partially transparent. This powerful notion

[32] Such as: An image is a 0D, 1D, 2D, 3D, or 4D rectilinear array of pixels where a pixel consists of one or more channelvalues of identical type, all representing color field component samples, except with perhaps one representing an opacity field sample.

[33] Or equivalently, to be completely transparent. I have never dealt with this possibility in practice, but it can be conceived so is a potential problem. The pixel object might have to be augmented with a flag this special case in case of the problem.

[34] We use a real number for alpha between 0. and 1. for convenience only. It has often been realized in an 8-bit unsigned byte with values between 0 and 255.

allows arbitrary shapes including disconnected components and components with holes. The following definition captures the sprite theory notion of shaped image:

> A sprite *is an image with an alpha channel where the non-alpha channels are assumed to have been premultiplied by the alpha channel.*

A consequence of this definition is that no channelvalue in a pixel's non-alpha channels can exceed the channelvalue in the pixel's alpha channel.[35] In particular, transparent pixels (alpha is 0) have all non-alpha channelvalues 0 also. Such a pixel is referred to as *clear*. As already argued however, it is really a pixel that doesn't count anymore—that conceptually no longer exists. Some very efficient data representation scheme for the object might not even allocate memory space for clear pixels. In fact, to refer to clear pixels at all is problematical but reflects the reality that sprites *are* nearly always represented in rectilinear arrays that *do* "realize" clear pixels.

One can use more general images or non-premultiplied alphas when necessary or convenient—as does Altamira COMPOSER—but the most important object is the sprite. The particular sprite that will be used here for explanations is this not surprising restriction of the general definition:

> An (RGBA) sprite *is an (RGBA) image where the RGB channelvalues are assumed to have been premultiplied by the alpha A channel.*

It is convenient to talk about the upper-left pixel of a sprite, just as we do of an image. Of course, the upper-left pixel might not exist, in the sense that it has an alpha of 0. In that case, the upper-left pixel is that which would exist at the upper left if the support of the sprite were extended to a rectilinear set of points just including the support of the sprite. In fact, the most common way sprites have been represented so far to my knowledge is with rectilinear arrays of points just holding the

[35] This follows from the sprite theory definition of pixel which requires that all channels have channelvalues of the same type. If we allowed the alpha channel to be an exception to this rule and let it be a **float** with value α on [0.,1.], then a (premultiplied) non-alpha channelvalue C cannot exceed αC_{max}, where C_{max} is the maximum channelvalue C may attain.

sprite, for which case the upper-left pixel is well-defined, as a pixel address anyway. Again, the reality of application writing usually "realizes" the clear pixels of a sprite, so we shall talk about them for convenience as if they exist.

Definition of Shape

Sprites have shape. That is their distinction. The basic notion of the shape of an image with an alpha channel is Hume's "extension" of the image, being the subset of its pixels with non-0 alpha.[36] For a sprite this is simply the following:

> *The* (hard) shape *of a sprite is exactly the support of the sprite.*[37]

An interesting non-Humean variation on the definition is this:

> *The* (soft) shape *of a sprite is exactly the alpha channel of the sprite.*

This notion of shape includes not only extension (support) but opacity (alpha), or a measure of "how much" occupancy of space a pixel has. I will usually imply the soft shape when I use the word "shape."

It should not be forgotten that our definitions are meant to apply to temporal *duration* as well as spatial extension. So a (2+1)D temporal sequence of digital sprites has a space-time shape—either simple space-time occupancy in case of hard shape, or a weighted occupancy of space-time in case of soft shape. I believe this world of shaped space-time sprites is an interesting one awaiting serious exploration.

Thus the shape of a sprite (or an image in general) is defined in sampling terms, not geometrical. It is sometimes convenient—for example, succinct—to describe a shape geometrically (with a "region of interest" or a "domain of definition"), but what is *always* meant in sprite theory is this: The geometric "shape" is rendered into an image in an alpha chan-

[36] Hume apparently did not take complete transparency into account when discussing the extension of an object. Surely the notion of extension must include the space occupied by a completely colorless (and transparent) object. If so, then Hume's notion of extension and our notion of non-0 alpha are *not* synonymous.

[37] So the quotation at the beginning of this lecture should have been introduced with this: "Hume on extension (and alpha support)."

nel, at which point it becomes a shape by our definition. Again, the geometrical is kept cleanly distinct from the sampled.

Coordinate Systems

The notion of coordinate system usually comes equipped with real space connotations when applied to images, but in sprite theory an image does not have a real coordinate system. As usual, the continuous and discrete worlds are held carefully apart. Instead, the pixel "coordinates" of an image are simply the corresponding integer array indices. This is easy and already standardized. Nearly all modern programming languages use 0-based indices. The upper left pixel in an image thus has indices [0][0].[38] Its horizontal index increases to the right; its vertical index increases down.

The notion of a real coordinate system is often useful in an imaging application. For example, the creative space of Altamira COMPOSER is a 2D continuous, unbounded space with positive and negative real coordinates in both dimensions. In this application, a set of sprites can each be arbitrarily located in this space, so we must specify the mapping of a sprite's integer array indices to the real coordinates of the space. Altamira COMPOSER uses a creative space coordinate system that makes the mapping of sprites to it almost trivial. Its horizontal, x, axis increases to the right; its vertical, y, axis increases down. Then a sprite can be positioned at any integer coordinate pair in the space by simply mapping its upper left pixel (with indices [0][0]) to that integer pair. The pixels of an image or sprite fall always on points with integer coordinates.

The important point is that an image has lost the notion of any coordinate system that might have existed in the continuous entity that was sampled to yield the image. It is just a matrix. Any coordinate system associated with it has to be defined by an explicit mapping, and such a mapping is external to the image or sprite itself.

You will notice that I have begun to alternate between the words "image" and "sprite." From here on, I will tend to use "sprite" most fre-

[38] Using C-like array notation, with the left coordinate denoting row number, hence the vertical dimension. This would be (0, 0) in most other matrix address notations, the left coordinate being in the horizontal dimension.

quently, because an image is just a special case of a sprite. As already indicated, I will also talk about the upper left coordinates of a sprite although the sprite might not actually exist there.

Altamira COMPOSER also has a notion of depth priority for its sprites, in the sense that sprites can lie in front of, or overlapping, other sprites. In other words, there is a front-to-back ordering of any set of sprites. Although there really is no third creative space dimension in Altamira COMPOSER, it is sometimes convenient to talk as if there were one, called *z*, that increases away from the plane as a viewer might observe it to "see" the sprites. (We have not yet talked about actually displaying the sprites, so this is an abstract viewer.) Notice that the 3D space implied by the third coordinate is a right-handed coordinate system. There is no requirement in our imaging model that this real coordinate system be used in an application, but it is a remarkably simple one.[39] There is a requirement, however, that images and sprites be thought of *computationally* as arrays and indexed in the standard way—even if not actually stored that way. Of course, sprites are to be thought of *intuitively* as shaped images with nonexistent clear pixels.

Continuous Operators on Discrete Sprites

There are two ways to slide a sprite around in creative space. As usual, the two conceptions come from the two worlds, continuous and discrete, and I shall carefully distinguish them. One way is to *move* a sprite to a new location. This means to reassign its upper-left pixel to a new point with integer coordinates. This is the default action a user of Altamira COMPOSER gets when he or she clicks on the displayed representation of a sprite and drags it to a new position. The corresponding sprite is assigned a new location in the creative space (and also displayed in a new location in display space, which I have not yet detailed—see page 69).

The other way is to *translate* it to an arbitrary real location. As so simply stated, this action does not make any sense. Sprites aren't defined on real locations. Yet this is the usual kind of statement one often hears

[39] It is also very familiar since it is the natural space of the written word for most, if not all, Indo-European languages. One reads from left to right, from top to bottom, and from front to back.

about images with no further explanation, as if it were obvious what is meant. So here is our first example of the continuous model we associate with our discrete sprite model and to which continuous operators are applied. Detailing the translation operator will explain exactly how to think of continuous operators applied to discrete images and sprites. Our model derives directly from sampling theory. Figure 2 shows how to think of a continuous operator on a discrete sprite:

First, a sprite is reconstructed into a continuous object by applying a reconstruction filter to its samples, as the Sampling Theorem instructs us to do. Then the continuous operator is applied to the continuous object so obtained. Then the result of the operation, a new continuous object, is resampled by another application of the Sampling Theorem into a new sprite. For example, a translation of a sprite is performed by reconstructing the sprite, translating the continuous entity so obtained by the desired amount, and then resampling at the integers. The alpha channel is reconstructed, translated, and resampled too.[40] Now that we know exactly how translation is applied to a sprite, we can safely speak of "translating a sprite." And this model can be extended straightforwardly to the formula *reconstruct-transform-resample* to model any continuous transformation of an image or sprite.

Technically speaking, the transformed continuous entity should be low-pass filtered before resampling to remove any high frequencies contributed by the transform step—to satisfy the Sampling Theorem.[41] Often, as in translation or magnification, no high frequencies can be introduced, so the filtering step can be skipped. We should always be aware that the Sampling Theorem requires removal of frequencies greater than half the sampling rate before sampling can be performed accurately. So strictly speaking the formula is *reconstruct-transform-filter*-resample*, where *filter** means "filter if you need to." I find this a bit clumsy so will use the abbreviated form, so long as you remember that it is an abbreviation.

[40] One of the beauties of the premultiplied alpha representation is that all channels are treated identically. See appendix B.

[41] "Low-pass" filtering simply means that all frequencies below a designated "cutoff" frequency are allowed and all frequencies above it disallowed. The cutoff frequency for Sampling Theorem application is half the sampling rate.

It is important to notice what I have *not* said. I have not ever used the word "rectangle." In fact, the edge of a reconstructed sprite is, in general, not accurately represented by a rectangle. See Figure 2 for details. I have not ever said what the "shape" of a pixel is. We know by now that this does not make sense. Notice that if any continuous model of a pixel is to be defined, it should probably be intimately related to the reconstruction filter being used—in general, not a simple shape. I have not specified what the reconstruction filter is.[42] The correct one, according to the Sampling Theorem, is the sinc filter,[43] but this is infinite in spatial extent and thus not practical to use. In practice, a variety of approximations to the sinc filter, with finite spatial extent, are used. The worst approximation used is the simple box filter (source of the infamous, and inaccurate, little square model of the pixel). Altamira COMPOSER generally uses much more sophisticated cubic filters.[44] The filter used is an implementation detail, but I highly discourage the use of box filters—*ever*—to ensure high quality results. It should probably go without saying, but I will anyway: It is an error to use no reconstruction filter at all.

The Acid Test, or Final Exam

The following problem was proposed to me by an email correspondent.[45] If you solve it correctly, then I have probably conveyed to you the fundamentals of sprite theory, and why there should be a sprite theory. Here is the test, as originally presented to me. I applied the theory and was led immediately to the right result. See if you can. This is the kind of problem the theory is meant to resolve:

> If I scale a 2×2 pixels floating point image whose values are $(0,0,0,1)$, $(1,0,0,1)$, $(0,1,0,1)$, $(1,1,0,1)$—by $1024\times$—what is the result?

[42] The class of reconstruction filters represented by the footprint of Figure 2b is a simple one. The reconstruction filters of Altamira COMPOSER, for example, are generally bicubic which means that their footprint extends across two sampling intervals in each direction, not just one as shown in the figure. See Figure 3. So filters generally overlap a great deal.

[43] Recall that the $\mathrm{sinc}(x)$ filter is generated from the function of form $(\sin x)/x$.

[44] My favorite reconstruction filter is the bicubic filter formed from the Catmull-Rom cubic spline basis function in each dimension (Smith 1983).

[45] Pierre Jasmin.

SPRITE THEORY

In sprite theory words: Consider a image of size $(2, 2)$ where the four opaque RGBA pixels are black and red in the top row, and green and cyan in the bottom row. Scale this image by 1024. What image do you get?

If your result is an image of four large squares of solid colors, the same four solid colors as the four given pixels have, then you have failed the test. You are stuck in geometry, have confused it with imaging, and have not correctly understood sampling. What you have done is to magnify the geometric image of four squares of the given colors in real continuous space. Then you resampled (without filtering for high frequencies—a sampling theory error) into an image.[46] The "scaling" in this case is actually just simple pixel replication, not true magnification.

Here's how to apply sprite theory: One of the fundamental assumptions is that an image represents a correctly sampled continuum. To perform a true magnify by 1024 then you must reconstruct, transform, and resample. So, using a good filter of course (not a box, in particular), you would proceed to place a copy of it at each pixel, weighted by the values there. Then you would add up all the contributions of the weighted filters wherever they were non-0 to get the final reconstruction. I am describing this in words because the continuum you get depends on the reconstruction filter you use. If the filter has support $\{-2., 2.\}$ in each dimension, as is typical for good reconstruction, then the reconstruction will be non-0 over the range $\{-2., 4.\}$ in general in each dimension. In general it will not be a square or a rectangle (see Figure 2) and it will not consist of solid colors, but rather ramps of color between the given four values. Now this continuous object is scaled by 1024 in each dimension. Since it being scaled up in size, no high frequencies are introduced (in fact, all frequencies are made lower since they are spread over greater distances), so no low-pass filtering is required. Now the larger, continuous object of ramped colors and sloping edges is resampled at all integer points where the object is non-0. This is the sprite theory (or sam-

[46] The intuition you need here is this: A hard edge in continuous space requires extremely (infinitely) high frequencies to represent in the frequency domain. Four geometric squares of solid color are separated by such hard edges. One would know that high frequencies were present, intuitively, and that therefore low-pass filtering must be applied before resampling. This filtering would have the effect of softening the transition from one color into the neighboring one.

pling theory) result. It has no areas of constant color. It is wider and taller than the original (2, 2) image. There is never any question, given the theory, of what the outcome has to be. There is never any question, given the theory, about where the output pixels "are located." There was no geometry at any imaging step, because we rigorously kept the discrete distinct from the continuous. The only variation allowed is the natural one that depends on the shape of the reconstruction filter used. Thus the exact filter used is important data to maintain and convey to others.

If the theory is widely adopted, then no two applications should produce different results, given the same reconstruction filter, as is now the case. There should be no slight shifts between results (a coordinate system inconsistency), or grossly different outcomes (the sampling-geometry confusion). As I have said, I have seen all these errors, even within a single organization.

BOX ALGEBRA

Box Algebra

Support calculations are common in image computation, just as address or pointer calculations are common in ordinary computation. Sprite theory provides a rich arithmetic, called the *box algebra*,[47] for support calculations.

By sprite theory convention, the pixels of an image or sprite always reside on the integer grid. Each point of the nD integer grid is described by an n-tuple of integer coordinates, an nD integer *point*. The integer point associated with a pixel in an image can be thought of as its address in integer coordinate space—its *pixel address*—not to be confused with a physical memory location allocated to hold the pixel. The *support* of an image is the set of pixel addresses of its pixels, and the support of a sprite is the set of its non-clear pixel addresses. The rectilinear support of an image or sprite can be represented by its minimally enclosing *box*. We again carefully distinguish a discrete concept (box) from a geometrical one (rectangle). A box is to be thought of as a (rectilinear) bag of pixels—the support of the pixels actually—not as a rectangle.

Rectangles—real geometrical rectangles—are often handy, so points and boxes with real coordinates are defined to represent them. As usual in digital computation, we will use a floating-point type variable to approximate real numbers. That type variable will be said to be of **float** type.

This box algebra consists of arithmetic and set operations, extended to points and boxes, and new operators for box validity, intersection, and construction. Assignment is extended to points and boxes. The intimate relationship between the two is spelled out.

[47] I attempt to justify the use of the word "algebra" in a later section, page 45.

BOX ALGEBRA

Support

When an image is declared, its pixels are located at all nD integer points in the rectilinear set of points extending from $\mathbf{0} \equiv (0, 0, \ldots, 0)$ to $(size_0 - 1, size_1 - 1, \ldots, size_n - 1)$, where $size_i$ is the number of pixels of the image in dimension i. Any rectilinear subset of an image is also an image, or *subimage*. Subimages can reside anywhere within an allocated image so may have arbitrary nonnegative coordinates in the range of coordinates spanned by the image. The rectilinear set of integer points which index the pixels of an image or subimage form its support. Similarly for sprite and *subsprite*, but remembering in this case that clear pixels are intuitively nonexistent.

Points

Image computations frequently refer to image support points or to reference points within images. Points are also often used to express offsets, in the sense of a vector relative the origin of a coordinate system. Sprite theory provides two formal data objects for points, called **point** and **floatpoint**, for points with integer and floating-point coordinates, respectively. Henceforth, we restrict ourselves to the 2D case, generalization to higher dimension not being difficult. As a practical example, Altamira COMPOSER implemented only the 2D case.

A **point** has two elements, called *coordinates*, of type **int**.[48] A **floatpoint** has two coordinates of type **float**. The coordinates are assumed to be named x and y. For convenience, given a **point** or **floatpoint** p, its coordinates are referenced as $p.x$ and $p.y$. In the actual implementation of Altamira COMPOSER, there are methods used for setting or returning these coordinates, but it is more succinct in documents such as this to use the '.' notation. We shall call this an example of the *meta-notation*, which tends to look better in print than the typical programming language.

There is a special **point** called **zeropoint** always available that is simply a point with two 0 coordinates. Altamira COMPOSER implements this

[48] Undefined types are assumed to be C-like types.

as a method **point** ZeroPoint(**void**). A useful method is **boolean** IsZeroPoint(**point** p). A **boolean**, naturally, has two values **true** or **false**.

Another very useful method is **boolean** EqualPoint(**point** p, **point** q) that is **true** if p is exactly equal to q coordinatewise. We do *not* represent this in meta-notation with $p \equiv q$. See the relational operators below to discover that $p \equiv q$ is meta-notation for **booleanpoint** EQPoint(**point** p, **point** q), returning a **booleanpoint** rather than a **boolean**. I will return to this again, after I introduce the type **booleanpoint**, although its definition will probably not come as a surprise.

A useful point operator in 2D is one that simply exchanges the coordinates of a point. In Altamira COMPOSER the method is **point** TransposePoint(**point** p).

A **point** can be cast to a **floatpoint**, and vice versa. The action is the one expected by anyone familiar with C: **int**s are cast as **float**s, coordinate by coordinate, or the reverse, in which case truncation is performed. In the Altamira COMPOSER implementation, these casts are performed by explicit methods **point** IntPoint(**floatpoint** p) and **floatpoint** FloatPoint(**point** p). The meta-notation here is $p = q$. Another conversion supplied is **point** RoundIntPoint(**floatpoint** p) that rounds up each coordinate rather than truncating it.

For any **floatpoint** p, there are always two interesting **point**s available, the *underpoint* and the *overpoint*. The underpoint of p is the **point** with each coordinate equal to the integer just less than or equal to—in the sense of the **floor**( ) function—the corresponding coordinate of p. Looking at the real line oriented left to right with the positive reals increasing toward the right, this is the integer just left of (or equal to) the given coordinate. Similarly, the overpoint is the point with each coordinate equal to the integer just greater than or equal to it—in the sense of the **ceil**( ) function—that is, the integer just right of (or equal to) the coordinate on the real line as just described. Meta-notation for these points are $p.under$ and $p.over$, or equivalently, $\lfloor p \rfloor$ and $\lceil p \rceil$. Altamira COMPOSER does not implement the methods for these, which might be, say, **point** UnderPoint(**floatpoint** p) and **point** OverPoint(**floatpoint** p).

BOX ALGEBRA

The following arithmetic operators are defined for **point**s. I give a meta-notation, the Altamira COMPOSER implementation method, and a C++ definition for each. Clearly, if Altamira COMPOSER had been implemented in C++, handy use of overloaded operators could have been used,[49] as indicated by the meta-notation.

$p + q$	**point** AddPoint(**point** p, **point** q)	+ coordinate-wise
$p - q$	**point** SubPoint(**point** p, **point** q)	– coordinate-wise
$-p$	**point** MinusPoint(**point** p)	– coordinate-wise
$p \,\%\, mod$	**point** ModPoint(**point** p, **point** mod)	% coordinate-wise
$p * q$	**point** MulPoint(**point** p, **point** q)	* coordinate-wise
p / q	**point** DivPoint(**point** p, **point** q)	/ coordinate-wise

For **floatpoint**s:

$p * q$	**floatpoint** MulFloatPoint(**floatpoint** p, **floatpoint** q)	* coordinate-wise
p / q	**floatpoint** DivFloatPoint(**floatpoint** p, **floatpoint** q)	/ coordinate-wise

A *mask* is defined to be an **int** with bits set to 0 or 1.

We define a **booleanpoint** to be a **point** with **boolean** coordinates. There is an obvious casting allowed between **booleanpoint** and **point**. In Altamira COMPOSER we have elected to simply use a **point** with coordinates values of 0 or 1 to represent a **booleanpoint** to avoid proliferation of types.

An example of the use of a mask is the method **int** PointToMask(**point** p) that converts a **point** to a mask as follows: Bit 0 of the mask is set to 1 if $p.x$ is non-0, and bit 1 is set if $p.y$ is non-0. This is particularly useful if p represents a **booleanpoint**. Conversely, **point** MaskToPoint(**int** m) converts a mask to a **point** (cf., **booleanpoint**).

The following relational operators return **booleanpoint**s:

$p \equiv q$	**booleanpoint** EQPoint(**point** p, **point** q)	== coordinate-wise
$p \neq q$	**booleanpoint** NEPoint(**point** p, **point** q)	!= coordinate-wise
$p < q$	**booleanpoint** LTPoint(**point** p, **point** q)	< coordinate-wise
$p \leq q$	**booleanpoint** LEPoint(**point** p, **point** q)	<= coordinate-wise
$p > q$	**booleanpoint** GTPoint(**point** p, **point** q)	> coordinate-wise
$p \geq q$	**booleanpoint** GEPoint(**point** p, **point** q)	>= coordinate-wise
$m\,?\,p : q$	**booleanpoint** WherePoint(**booleanpoint** m, **point** p,	?: coordinate-wise

[49] I realize that such overloading is frowned upon, but I couldn't resist it when I implemented an early version of ICEMAN.

	point q)	

Boxes

The most common support structure required in imaging is the box. Sometimes an entire image computation can be performed on its boxes alone, without ever referring to the actual pixel values of the image except for trivial data copying. See An Informative Example, page 59.

The model provides two formal data objects for boxes, called **box** and **floatbox**, for boxes with integer and floating-point coordinates, respectively. There are several ways to implement a **box** or **floatbox**. I shall not dictate an implementation but require that several characteristics of them should always be available.

For example, a box (**box** or **floatbox**) has *minmax variables*. These are **int**s called *xmin*, *xmax*, *ymin*, *ymax*. They have the obvious meaning of the smallest and largest coordinate in each dimension represented by a box. Altamira COMPOSER has methods for retrieving minmax variables. For example, **int** BoxXMax(**box** b) returns *xmax* of *b*, or $b.xmax$ in meta-notation.

Every box has a *minpoint* and a *maxpoint*, that are the points (**point** or **floatpoint**) corresponding to (*xmin*, *ymin*) and (*xmax*, *ymax*), respectively. Altamira COMPOSER has methods, such as **point** BoxMaxPoint(**box** b), for retrieving these. $b.max$ and $b.min$ are the maxpoint and minpoint of box *b* in meta-notation, or equivalently, $\bar{b}$ and $\underline{b}$.

Every box has a *size* that is a point. The definition of size differs for **box** and **floatbox**, however. The horizontal size of a **box** is $xmax - xmin + 1$, but for a **floatbox** it is $xmax - xmin$. The difference reflects the purpose of the two types. A **box** represents the support of an image. Its size is the number of pixels across and down the image supported. But a **floatbox** usually represents a geometrical rectangle, so its size is the actual real width and height of the rectangle represented. The pertinent Altamira COMPOSER methods here are **point** BoxSize(**box** b) and **floatpoint** FloatBoxSize(**floatbox** b). $b.size$ is the size (point) of box *b* in meta-notation.

BOX ALGEBRA

A **box** can be *invalid*—if its *xmin* is greater than its *xmax* or its *ymin* is greater than its *ymax*. So there is a method **boolean** ValidBox(**box** b) that determines box validity. For completeness, there would be a similar method for **floatbox**, but we never found a need for it in Altamira COMPOSER.[50] There is also the notion of **invalidbox**, a **box** guaranteed to be invalid. Notice that a **box** may have its minpoint equal to its maxpoint. This represents an image consisting of a single pixel. A **floatbox** b with $b.min \equiv b.max$—that is, a "rectangle" consisting of a single point—is considered a valid rectangle.

The equality of two boxes is determined with **boolean** EqualBox(**box** b, **box** c)[51] that is **true** if b and c represent exactly the same set of pixels—that is, have the same box.

Boxes can be constructed in a variety of useful ways, summarized, for **box**, by the list of construction methods below. There is a similar list for **floatbox**.

box BoxConstruct(**int** xmin, **int** xmax, **int** ymin, **int** ymax)
box BoxOriginSize(**point** origin, **point** size)
box BoxFromPoints(**point** p, **point** q)
box BoxAbsFromPoints(**point** p, **point** q)

The first two of these are self-explanatory. The third constructor creates a box that minimally includes p and q. So p and q must lie at either end of a diagonal of the box, thinking of it as a rectangle. It is convenient to have meta-notation for the operator defined by this constructor; it is $p \sqcup q$, and $\sqcup$ is called the *box operator*.[52] The fourth one assumes p and q are the minpoint and maxpoint, respectively, of the box constructed. In Altamira COMPOSER it can create an invalid box since no checking is done.

[50] In fact, very little functionality for **floatpoint**s and **floatbox**s was implemented in Altamira COMPOSER. We implemented just those methods we actually needed.
[51] Altamira COMPOSER actually implements many **boolean**s as **int**s, but this is a minor implementation detail.
[52] The fact that this operator looks like a "boxy" version of the classic set union operator ∪ is not a coincidence. Note, however, that the box operator is not a set union but generally creates a set larger than the set union of its arguments.

A box has several extremal points, called *corners*. Its minpoint and maxpoint are two of these. Method **point** BoxCorner(**box** b, **int** mask) returns the corner of box *b* specified by the bitmask *mask*, where the low-order bit represents the *x* dimension and the next-higher bit *y*. For example, *mask* set to value b01 yields the upper right corner.[53] Meta-notation for this corner is $b.corner[b01]$, or equivalently $b.corner[1]$.

The *center* of a box, thought of as a rectangle, is obtained with is obtained with a method **point** BoxCenter(**box** b), which truncates, or **floatpoint** FloatBoxCenter(**box** b), which rounds. Meta-notation is $b.center$.

With every **box** is associated another called its *basebox* which is the box moved to the origin—so that its minpoint is the origin **0**. This is $b.base$ in meta-notation, and the Altamira COMPOSER method is **box** BaseBox(**box** b). The corresponding notion for **floatbox** was not implemented.

A **box** may be cast to a **floatbox** and vice versa. In Altamira COMPOSER, the methods are **box** IntBox(**floatbox** b), which truncates, **box** RoundIntBox(**floatbox** b), which rounds, and **floatbox** FloatBox(**box** b), which does neither.

Two very useful notions for **floatbox**s are those of *innerbox* and *outerbox*. The innerbox of **floatbox** *b* is the **box** on the integers just inside or on the given box. The outerbox is the **box** on the integers just outside or on the given box. The Altamira COMPOSER methods are **box** InnerBox(**floatbox** b) and **box** OuterBox(**floatbox** b). In meta-notation, $b.inner$ and $b.outer$ represent these two boxes, respectively, or equivalently $\lfloor b \rfloor$ and $\lceil b \rceil$. In meta-notation, the definitions are

$$\lfloor b \rfloor \stackrel{\text{def}}{=} \lfloor \underline{b} \rfloor \sqcup \lfloor \overline{b} \rfloor, \text{or } b.inner \stackrel{\text{def}}{=} (b.min).over \sqcup (b.max).under$$

and

$$\lceil b \rceil \stackrel{\text{def}}{=} \lfloor \underline{b} \rfloor \sqcup \lceil \overline{b} \rceil, \text{or } b.outer \stackrel{\text{def}}{=} (b.min).under \sqcup (b.max).over.$$

[53] The "b" in "b01" indicates that the value "01" is expressed in binary notation.

BOX ALGEBRA

Box Operators

Two handy unary **box** operators are **box** TransposeBox(**box** b) and **box** RightRotateBox(**box** b). TransposeBox( ) swaps the horizontal and vertical sizes of a **box** while leaving the minpoint fixed. RightRotateBox( ) returns a **box** that is approximately what you would get if you thought of *b* as a rectangle and rotated it about its center. This is a good place to show the box algebra in action with the code for implementing RightRotateBox( ). This is the actual code from Altamira COMPOSER:

```
box RightRotateBox(box b) {
    point ptoffset = SubPoint(BoxCenter(b), MinPoint(b));
    point ptmin = SubPoint(BoxCenter(b), TransposePoint(ptoffset));
    return BoxOriginSize(ptmin, TransposePoint(BoxSize(b)));
}
```

The set of binary **box** operators is given below:

```
box IntersectBox(box b, box c)
box UnionBox(box b, box c)
box BoxPlusBox(box b, box c)
```

IntersectBox( ) returns the **box** representing the geometric intersection of its two arguments, treated as geometric rectangles. The result can be an invalid box in the case of a complete *miss*—that is, the two boxes don't intersect. The intersection *a* of box *b* and box *c* is computed in the *x* dimension by

```
a.xmin = c.xmin < b.xmin ? b.xmin : c.xmin
a.xmax = c.xmax > b.xmax ? b.xmax : c.xmax
```

and similarly for *y*. Meta-notation for intersection is $b \sqcap c$, where $\sqcap$ is called the *intersection operator*.[54] Let $w \equiv b \sqcap c$, then meta-notation lets us express the calculation above another way: The minimum of each dimension i (x or y) is given by

$$\underline{w}.i \equiv \max(\underline{b}.i, \underline{c}.i)$$

[54] The fact that this operator looks like a "boxy" version of the classic set intersection operator ∩ is not a coincidence.

and the maximum by

$$\overline{w}.i \equiv min(\overline{b}.i, \overline{c}.i).$$

The **floatbox** version of intersection was not implemented in Altamira COMPOSER.

UnionBox( ) returns the minimal **box** enclosing the two **box** operands, meta-notation for which is $b \sqcup c$, using the box operator introduced earlier for construction of a box from two points. This use is consistent with the former if a **point** is taken to be a degenerate **box**. In meta-notation let $w \equiv b \sqcup c$, then the calculation defining the union is, for each dimension i (x or y):

$$\underline{w}.i \equiv \min(\underline{b}.i, \underline{c}.i)$$
$$\overline{w}.i \equiv max(\overline{b}.i, \overline{c}.i).$$

The **floatbox** version was not implemented in Altamira COMPOSER.

BoxPlusBox( ), or $b + c$ in meta-notation, is defined by

```
BoxFromPoints( AddPoint(MinPoint(b), MinPoint(c)),
    AddPoint(MaxPoint(b), MaxPoint(c)))
```

or by

$$b + c \stackrel{\text{def}}{=} \left(\underline{b} + \underline{c}\right) \sqcup \left(\overline{b} + \overline{c}\right)$$

in meta-notation. It is most useful and understandable when the minpoint of c is completely negative and the maxpoint is positive. Then the result is seen to be a **box** that one would get by taking the union of b with all possible positions of c such that the origin of c is in or on b. Another way to think of it in this case is that b is expanded by c. The **floatbox** version is defined similarly.

The following **boolean** functions on boxes are defined:

boolean BoxInBox(**box** b, **box** c)

that returns **true** if b lies totally within c, both treated as rectangles. b may share an edge with c and still give **true**. Meta-notation is $b \subseteq c$.

boolean PointInBox(point p, box b)

that returns **true** if p lies in or on b, treated as a rectangle. Meta-notation is $p \subseteq b$.

The following operators combine a box and a point:

box BoxPlusPoint(**box** b, **point** p)
box BoxMinusPoint(**box** b, **point** p)
floatbox MulFloatBox(**floatbox** b, **floatpoint** p)
box BoxExpand(**box** b, **point** p)

BoxPlusPoint( ) offsets **box** b by **point** p. Meta-notation is $b + p$. The definition in meta-notation is

$$b + p \stackrel{\text{def}}{=} (\underline{b} + p) \sqcup (\overline{b} + p).$$

The **floatbox** version is defined similarly.

In general, an operator **op** between two points can be extended to a box b and a point p by the form

$$b \textbf{ op } p \stackrel{\text{def}}{=} (\underline{b} \textbf{ op } p) \sqcup (\overline{b} \textbf{ op } p)$$

and to a box b and a box c by the form

$$b \textbf{ op } c \stackrel{\text{def}}{=} (\underline{b} \textbf{ op } \underline{c}) \sqcup (\overline{b} \textbf{ op } \overline{c}).$$

BoxMinusPoint( ), $b - p$, is defined similarly to $b + p$. The **floatbox** version was not implemented in Altamira COMPOSER.

MulFloatBox( ), $b * p$, is defined similarly. In this case, it was the **box** version that was not implemented. This routine is typically used to scale b by a size p.

BoxExpand( ) returns a box expanded (or shrunk) by adding p to $b.max$ and subtracting it from $b.min$. The **floatbox** version was not implemented.

A very useful method, NotBox( ), has this prototype in Altamira COMPOSER:

int NotBox(**box** b, **box** B, **box*** top, **box*** bottom, **box*** left, **box*** right)

that returns the complement of box b in (what is assumed to be) surrounding box B as four boxes: *top* and *bottom* are as wide as B; *left* and *right* are as high as b. The returned value is a 4-bit flag with one bit corresponding re-

spectively to each returned box. Each bit is 1 if the corresponding box is valid. Note that NotBox( ) does not define a useful complement operator for boxes, because it does not return a box, but rather a set of boxes.

Special Box Routines[55]

After programming with an early version of the concepts at Pixar, I noticed that I was solving the same two problems over and over again. This is described in detail in (Smith 1989). I invented two nonobvious functions that greatly eased this situation:

> **boolean** AlignSrcAndDstBoxesWithOffset(**box** s, **box** d, **point** p, **box*** S, **box*** D)

This useful routine takes an arbitrary input source box and input destination box (assumed to define source and destination subimages) and "aligns" them, where there may be an arbitrary offset between them. So the minpoints are aligned unless there is a nonzero offset, in which case the source box is aligned with its minpoint offset relative the minpoint of the destination box. The boxes and point must lie in the same coordinate system. The output source and destination boxes define the minimally affected subimages of the images defined by the input boxes. They represent the intersection of the two input boxes and are thus the same size. They are, however, generally different boxes since they are subboxes of an arbitrary pair of input boxes. Let s and d be the input boxes and S and D be output boxes. Let p be the offset. Then, in metanotation mixed with some C-like code, the computation is (t is a temporary **box** variable):

$t = (d.base - p) \sqcap s.base$;
if(!ValidBox(t)) **return false;**
$*S = t + s.min$;
$*D = t + d.min + p$;
return true;

[55] This section may be skipped until after motivations for the two routines are established in the Image Assignment section, page 57, of the next chapter.

The other useful routine is:

boolean AlignSrcAndDstSubBoxes(**box** s, **box** d, **box** b, **box*** S, **box*** D).

Given two aligned boxes (as, for example, output by the routine above) and given a box *b* which intersects the input source box *s* of the two, determine the intersection and the corresponding subbox of the input destination box *d*. Return these two subboxes, which are the same size, by definition. *S* and *D* are these two "aligned" subboxes of the given aligned boxes. The routine returns **true** if the subboxes are valid else **false**. If **false**, then *S* and *D* are undefined. *b* must be in the same coordinate space as *s*. Let *S* and *D* be the aligned input boxes. Let *s* and *d* be the aligned output boxes, if any. Then the computation in meta-notation is:

$t = S \sqcap b$;
if(!ValidBox(t)) **return false**;
$*S = t$;
$*D = t + D.min - S.min$;
return true;

The Algebraic Structure of Box Algebra[56]

I have called the calculus of boxes a box algebra. What kind of algebra is it? When I set out to write this section, I thought I could derive an algebra of boxes that was complete in the sense of an algebraic field: (1) It was closed on two binary operators, which were each commutative and associative, and which distributed over one another, and for each of which there existed an identity; and (2) there existed an inverse or complement for each operator. I will show (1) here, but not (2), a problem I will leave to the interested reader.[57]

I will show that ⊔ and ⊓ have the right properties for the additive and multiplicative operators of an algebraic field, and identity elements for them will be defined. The algebraic structure derived is rich enough to

[56] This section can be skipped with no substantive loss to the presentation of the theory. It is included for completeness.

[57] In an earlier version of this material, I defined the complement of a box to be the box obtained by swapping the minpoint and the maxpoint, interpreted as a box passing through infinity or around a toroidally connected space. Does this defintion work?

justify use of the term "algebra," but I could find no important reason for implementing it fully as described in this section.

Let the special **emptybox** be a box that represents enclosure of no space. Notice that a box could be empty in one dimension but not in another. Such a *partially empty* box is not **emptybox**.

Let another special box, **universalbox**, represent enclosure of all of 2D space. A box can be universal in one dimension but not another. Such a *partially universal* box is not **universalbox**.

Both **universalbox** and **emptybox** are considered valid. A *normal* box is any valid box other than **universalbox**, **emptybox**, a partially universal box, or a partially empty box. I will just assume that the implementation of **box** allows for the two special cases, without specifying how it is actually done. **universalbox** and **emptybox** will be shown to be the identities for the $\sqcap$ and $\sqcup$ operators, respectively.

The $\sqcup$ operator, the box operator, is commutative. That is, $b \sqcup c \equiv c \sqcup b$, for boxes b and c, because finding minima and maxima is not dependent on argument order. It is also idempotent, $b \sqcup b \equiv b$.

The $\sqcap$ operator is commutative by the same argument. Thus $b \sqcap c \equiv c \sqcap b$. It is also idempotent: $b \sqcap b \equiv b$.

The $\sqcup$ operator can be shown to be associative. The definition of $\sqcup$ applied to $w \equiv (a \sqcup b) \sqcup c$, for boxes w, a, b, and c, determines the minimum of each dimension i (x or y) to be

$$\underline{w}.i \equiv \min(\min(\underline{a}.i, \underline{b}.i), \underline{c}.i)$$

and the maximum to be

$$\overline{w}.i \equiv max(max(\overline{a}.i, \overline{b}.i), \overline{c}.i).$$

Since min and max are associative, $\sqcup$ is too.

The $\sqcap$ operator can also be shown to be associative. The definition of $\sqcap$ applied to $w \equiv (a \sqcap b) \sqcap c$, for boxes w, a, b, and c, determines the minimum of each dimension i (x or y) to be

$$\underline{w}.i \equiv \max(\max(\underline{a}.i, \underline{b}.i), \underline{c}.i)$$

and the maximum to be

$$\overline{w}.i \equiv min(min(\overline{a}.i, \overline{b}.i), \overline{c}.i).$$

Since min and max are associative, ⊓ is too.

It can also be shown that ⊓ is distributive over ⊔. Let $u \equiv a \sqcap (b \sqcup c)$ and $v \equiv (a \sqcap b) \sqcup (a \sqcap c)$. Then for each dimension i (x or y),

$$\underline{u}.i \equiv \max\left(\underline{a}.i, \min(\underline{b}.i, \underline{c}.i)\right)$$
$$\underline{v}.i \equiv \min\left(\max(\underline{a}.i, \underline{b}.i), \max(\underline{a}.i, \underline{c}.i)\right).$$

Without loss of generality, assume $\underline{b}.i < \underline{c}.i$. Then both $\underline{u}.i$ and $\underline{v}.i$ are $\max(\underline{a}.i, \underline{b}.i)$. A similar argument holds for $\overline{u}.i$ and $\overline{v}.i$, so $u \equiv v$. Similarly, ⊔ can be shown to be distributive over ⊓.

Finally, notice that $b \sqcap$ **universalbox** $\equiv b$, and $b \sqcup$ **emptybox** $\equiv b$, for any box b, so **universalbox** and **emptybox** are the identity elements for ⊓ and ⊔, respectively.

I consider the baggage required for representing **universalbox**, **emptybox**, a partially universal box, and a partially empty box not justified. **emptybox** could be used for a complete miss in box intersection, but **invalidbox** serves this purpose in the box algebra as implemented.

I will continue to refer to the "box algebra," knowing that we could, if necessary, extend it to the algebraic structure defined in this section, but not doing so for the reasons just given.

IMAGE ALGEBRA

Image Algebra

We come at last to the meat and potatoes of the theory: image computations. The purpose of this chapter is to extend the model to include precise descriptions of images and sprites, operations that can be performed on them, and—importantly—operations that can be performed between them. Just as the preceding chapter defined points and boxes and built up an algebra of operators on them for support calculations, this chapter defines channels, pixels, and images and an algebra[58] between them for image calculations. Since each image has a support box by definition, box algebra concepts are integral to the image algebra developed here. That's why we so carefully developed them in the last chapter.

Channels

As already discussed, a pixel in the model may have an arbitrary finite number of channels, where a pixel channel represents a single numerical sample of some continuum. Similarly, an image may have any number of channels. So the **channel** is a fundamental object in the model, which will be used to formally define pixels and images, hence sprites.

One of the simplifications of the Altamira model is to allow only one *channeltype* per image (hence pixel). Mixtures of different channeltype are handled at a higher level.[59] It is convenient to adopt for valid channeltypes those supported by a development environment. The model abstracts these to the following example types that are intended to map naturally to common data types:

unt*n* **int***n* **float***n*

where *n* is the number of bits, typically 8, 16, 32, etc. This list is not meant to be exhaustive; it is extensible at will. **unt** stands for unsigned integer

[58] I do not attempt to extend this "algebra" to a mathematically complete one, as we did for the box algebra. I use the word simply for symmetry.

[59] This higher level is called the "imagestruct" level in Altamira COMPOSER.

and **int** for signed integer. A **float** represents a floating-point number. Practically, Altamira COMPOSER maps **unt8** to C's **unsigned char** and used this one type for almost every image, sprite, and pixel. The other types that Altamira COMPOSER at least recognizes are **unt32**, **float32**, and **float64**.

The code, however, is written to handle **unt**s of arbitrary *typesize*, measured in bytes. For example, **unt8** has typesize 1. The Altamira COMPOSER implementation of the model provides the methods **int** ChannelSize(**channel**) and **int** ChannelType(**channel**) to access these basic characteristics of a channel. The former returns the byte count of the typesize of the given channel, and the latter returns an index into an enumeration of available channeltypes.

The other fundamental characteristic of a **channel** object is the number of channels it contains. This is called its *ply*. Altamira COMPOSER provides method **int** ChannelPly(**channel**) for this important number.

Given a **channel** *ch*, meta-notation *ch.size*, *ch.type*, and *ch.ply* represent the characteristics above. As will be seen, a **channel** is used to define an **image** (hence a **sprite**) and a **pixel**, so these objects will inherit the channel characteristics. The meta-notation is extended in the obvious way to *I.size*, *I.type*, and *I.ply* for **image** *I* and *px.size*, *px.type*, and *px.ply* for **pixel** *px*. Similarly Altamira COMPOSER provides corresponding methods with the names you might guess—for example, ImagePly( ) and PixelType( ).

Finally, to completely define a channel, a *permutation* must be specified. This is a mapping of the channels to themselves. The default permutation is the identity. Thus an image might have channels called R, G, B, and A as its natural order (identity permutation), but to interchange G and B channels, say, only the permutation would have to be changed. So data in an image or pixel channel is accessed via indirection through the channel permutation.

Methods are assumed to exist for testing equality of **channel**s and channeltypes.

Color

It is not necessary to discuss color before defining pixels and images with more care, but the definition of RGBA sprite tells us that color is an important topic in that subset of the theory. So we discuss here the color objects that are used with sprites.

There are two integer color objects, **rgbcolor** and **rgbacolor**, and two float color objects, **rgbfloatcolor** and **rgbafloatcolor**.[60] **rgb[a]color** holds RGB[A] tuples as integers, and **rgb[a]floatcolor** holds RGB[A] tuples as reals. Meta-notation is straightforward. For example, **rgbacolor** *rgba* has R, G, B, and A *components* referred to as *rgba.r*, *rgba.g*, *rgba.b*, and *rgba.a*, respectively. There are methods in Altamira COMPOSER, of course, for setting and retrieving color components to and from colors.

The semantics of these color tuples can be any 3D or 4D color desired. For example, Altamira COMPOSER used **rgbcolor** objects to hold Hwb (Hue, whiteness, blackness) and HSV (Hue, Saturation, Value) representations of color (Smith & Lyons 1996). The point is, the color objects do not have to hold RGB color representations, despite the name. There are color space conversion methods for converting between RGB and Hwb and between RGB and HSV—for example, **rgbcolor*** HwbToRGB(**rgbcolor*** phwb).[61]

These types can be converted to one another in the ways that you might suppose. **rgb[a]color** can be promoted directly to **rgb[a]floatcolor**. An example method in Altamira COMPOSER is **rgbafloatcolor** FloatRGBA(**rgbacolor** rgba). If an alpha value is required to complete a conversion, it must be provided as in **rgbacolor** RGBtoRGBA(**rgbcolor** rgb, **int** alpha).

rgb[a]floatcolor can be truncated or rounded to **rgb[a]color**. Example methods are **rgbacolor** IntRGBA(**rgbafloatcolor** rgba) and **rgbacolor** RoundIntRGBA(**rgbafloatcolor** rgba).

[60] Altamira COMPOSER actually uses names RGBColorType, RGBAColorType, RGBFloatColortype, and RGBAFloatColorType, respectively.

[61] These were not actually implemented in Altamira COMPOSER, but the app does contain the conversions indicated, in a different guise—that is, between pixels.

Pixels

An object of obvious usefulness in image computing is the **pixel**, that holds all channels of one pixel of an image. A **channel** object and data for each channel defines a pixel. In Altamira COMPOSER, the constructor is **pixel*** PixelConstruct(**int** ply, **int** type, **int*** permute) that indirectly constructs a channel from the three arguments.[62]

RGB and RGBA pixels are of special interest and utility in image computing. These are pixels that hold an RGB color without or with an alpha value, respectively. These are so handy that Altamira COMPOSER provides special constructors for them that essentially take a **channel** and an **rgb[a]color** and produce an appropriate pixel: **pixel*** RGB[A]PixelConstruct(**int** ply, **int** type, **int*** permute, **rgb[a]color** c).

Pixels may be assigned to one another. In meta-notation *px* = *qx* for two pixels *px* and *qx*. This is straightforward if both pixels have the same channel structure. The general assignment has to interpret different channeltypes and different plys. The interpretation I have selected is this: channeltypes convert in the usual C-like ways. The channel structure of *px*, the receiving pixel, is unchanged by an assignment. If the ply of *qx* exceeds that of *px*, then the channelvalues of *qx* are simply assigned in order to the channelvalues of *px*. If the ply of *px* exceeds that of *qx*, then after the channels of *qx* are depleted, the remaining channels of *px* are assigned the last channelvalue of *qx*. This accomplishes the following: If *qx* is a single-ply pixel holding the single **unt8** value 255, and *px* is a 3-ply **unt8** pixel, then *px* = *qx* puts a 255 in all three channels of *px*. Altamira COMPOSER realizes this assignment with the routine **void** Pixel_Pixel(**pixel*** px, **pixel*** qx), read "pixel gets pixel."

Pixels can be converted to colors and vice versa—*rgb* = *px*, or *px* = *rgb*—in meta-notation, for example, for **rgbcolor** *rgb* and **pixel** *px*, and similarly for RGBA. Care must be taken for pixels without the natural ply of three for **rgbcolor** or four for **rgbacolor**. In the Altamira COMPOSER implementation of the model, **pixel*** Pixel_RGBAColor(**pixel*** px, **rgbacolor** rgba) simply returns NULL if *px.ply* is not equal to four. Otherwise the

[62] For clarity, I omit another argument CompStruct* *pComp* that Altamira COMPOSER threads through all routines. I do this as a general rule in this lecture.

assignment happens as would be expected. In the other direction, **rgbacolor** RGBAColor_Pixel(**pixel*** px) copies as many channelvalues as there are available, in order, to corresponding components of an **rgbacolor**, and any remaining components are set to 0. Similarly for conversion between **pixel** and **rgbcolor**.

Altamira COMPOSER provides a full set of color conversion routines that operate on pixels rather than colors. These include **pixel*** RgbToHwbPixel(**pixel*** px) and **pixel*** HwbToRgbPixel(**pixel*** px) for conversions to and from RGB and Hwb, and similarly for RGB and HSV. All these routines return an error of NULL if the ply of *px* is less than three.

Images, Cards, and Sprites

At last, we come to the most important object of all in image computing. In part 1, I defined the image as a rectilinear array of pixels. This is the conceptual model of the image. The actual implementation may be quite different. In other words, in true object-oriented fashion, the **image** object implementation is not dictated to be an array of **pixel** objects, but it could be.[63]

There are two popular ways to implement an image consistent with the sprite theory model—*layered* or *interleaved*. The layered method allocates image memory channel-wise, the interleaved method pixel-wise. It is important to understand, however, that the model does not dictate either. In fact, there are numerous ways to represent an image, but these two classes of ways deserve further explication.

It is easier to explain the different storage methods by referring to a concrete example. We use the RGBA image as an example, because it is of particular interest in computer graphics. An example of a layered method of representing an RGBA image allots separate pieces of memory to hold channel R, channel G, channel B, and channel A. Thus it maintains four pointers to the four pieces of memory. These four pieces of memory are conceived of as lying in register above one another in layers.

[63] I use **image** as the name of the object here to be consistent with Altamira COMPOSER. If I were to start from scratch I would use **sprite** as the name of the object.

The principal advantage of this method is the ease with which the ply of an image can be changed—from single channel rectilinear monochrome images to 64-channel, say, spectral band satellite images with one channel per spectral band filter. Another advantage is that the separate pieces of channel memory do not have to be contiguous. The disadvantage is the need for four pointers and their management.

An example of an interleaved method would allocate one contiguous piece of memory that would hold R, G, B, and A of the first pixel, then R, G, B, and A of the second pixel and so forth. The principal advantage of this method is that all color components are available in the vicinity of a single pointer—within small fixed offsets. The disadvantage is the large block of continguous memory required, plus the inability to add or subtract channels without massive data movement.

The **image** object of our model does care which of these, if either is adopted for actual implementation. The **image** could be implemented, for example, with virtual tiles and multiple resolutions.

The **image** object constructor requires only a size and a **channel** object. For example, in Altamira COMPOSER the image constructor is essentially[64] **image*** ImageConstruct(**point** size, **channel*** pch, **int** flag). The minpoint of an image is always [0][0] (but see the discussion of subimage below). The size of the image—meta-notation: *I.size* for image *I*—gives the width in the *x* coordinate and the height in the *y*. The type, ply, and mask of an image are inherited from its channelobject, *I.channel* in meta-notation, as discussed in the preceding chapter. Likewise for the permutation of the channel object. Meta-notations *I.box*, *I.basebox*, and *I.minpoint* represent the obvious information about image *I*.

In the Altamira COMPOSER implementation, the *flag* argument to the constructor is used for a variety of things. For example, an image is by default a layered representation, but the *flag* may be used to make it interleaved. It is also used to indicate whether an image with an alpha channel is to be interpreted to have premultiplied alpha or not (the default being premultiplied, of course).

[64] As before, the CompStruct* *pComp* object that is threaded throughout Altamira COMPOSER is not shown for succinctness of presentation.

Some useful mask methods are, in the Altamira COMPOSER implementation, **booleanpixel*** ImageMask(**image*** I), that returns a pixel representing *I.mask*, channel by channel, **boolean** ImageMaskData(**image*** I, **int** ch), that returns the mask for the given channel of image *I*, and **int** ImageMaskInt(**image*** I), that returns a bitmask for all channels of *I*.

The model has the notion of the **emptyimage**. The Altamira COMPOSER implementation uses method **image*** EmptyImageConstruct(**void**) to construct **emptyimage**, which has no data and a size of **zeropoint** (0, 0). It just simply exists. Since it must respond to all image methods, the Altamira COMPOSER implementation arbitrarily has it return a ply of 1, a type of **unt8**, and the identity permutation. Method **boolean** ValidImage(**image*** I) checks to see if an image is the **emptyimage** or not. It is sufficient to check that the size is **zeropoint**.

Another very useful special image is called a *card*. A card is a constant image object—an **image** with every pixel identical to all others and of arbitrary size. It can therefore be represented very succinctly. The Altamira COMPOSER constructor is **image*** Card(**pixel*** px, **int** flag), where the given pixel defines the constant pixel. Method **boolean** ImageCard(**image*** I) checks to see if an image is a card or not.

An extremely interesting special case of an image object is the *sprite*. As defined in Chapter 1, a sprite is an RGBA **image**, where the alpha is assumed to be premultiplied—that is, the RGB color channels are assumed each to have been premultiplied by the corresponding A in the alpha channel. For all practical purposes, the pixels with zero alpha can be considered simply to not exist. It is not necessary to allocate any storage for them, although it is still common to do so. There should be a method **boolean** ImageSprite(**image*** I) that checks to see if an image is a sprite or not. Altamira COMPOSER does not implement this nor the sprite explicitly although nearly all image objects in the application are sprites.

Some methods intended more for (RGBA) sprites than general images follow:

boolean RGBAtPoint(**image*** I, **point** p, **rgbacolor** rgba) returns the RGB color at point *p* of sprite *I*. This routine returns **false** if *p* is not on or within *I.box*. It returns black—that is, (0,0,0)—if *p* is within *I.box* but is a clear (nonexistent) pixel. Altamira COMPOSER returns **true** in this case, but an implementation might choose to return **false** instead. Similarly **boolean** AlphaAtPoint(**image*** I, **point** p, **rgbacolor** rgba) returns the alpha A at the point.

image* Card_RGBA(**rgbcolor** rgba) constructs a card sprite of the given color and alpha.

A useful notion for an **image** object is that of its *bounding box*. This is the minimal bounding box that contains pixels unequal to a given pixel. Usually the given pixel is clear (all 0s) and the bounding box then delineates those pixels that are "interesting"—that is, that have non-0 information in them. In the case of sprites, all pixels outside the bounding box (*bbox*, for short, pronounced "bee-box") may be discarded if memory space has been allocated for them. In Altamira COMPOSER the method is **box** ImageBoundingBox(**image*** I, **pixel*** px). The box returned is relative the box of the given image. It is **invalidbox** if all pixels equal the given pixel.

Subimages and Subsprites

A powerful notion of the model is that of *subimage*—and hence of *subsprite*. A subimage is an image that is a subset of another image. Thus a subimage is a rectilinear subset of the set of pixels in a given image. And a sprite is a rectilinearly bounded subset of pixels that are defined in a given sprite.[65] The important point is that a subimage (subsprite) is not separately allocated memory. Its memory is that used by the *parent* image (sprite). A major distinction is that its minpoint does not have to be **zeropoint**. A subimage is a way to focus attention within an image. But a subimage is an **image** so far as any image method is concerned. So, for example, a subimage may have subimages. A subimage is a *child*, of course, to its parent image or subimage.

[65] I believe the sprite version of everything I say about images is obvious so will cease spelling out its definition separately.

SPRITE THEORY

Our model always specifies the subimage of an image *relative* to the parent image. The Altamira COMPOSER method is **image*** SubImage(**image*** I, **box** b). It works like this: First, an error (NULL for Altamira COMPOSER) is returned if either *I* does not exist or *b* is invalid. Second, *b* is intersected with *I.basebox* to delineate the subimage pixels. In other words, *b* is cropped,[66] if necessary, to the given image. Then a new **image** object is allocated that inherits most characteristics from the given image, but has the size of the given box (cropped, if necessary), is marked a subimage, and refers to the data in the parent image rather than allocating its own. Its minpoint is set to the parent's minpoint plus the minpoint of the given box (again, cropped, if necessary).

Since a subimage is an image, all pixel references in it are relative its upper left pixel. The only exception is its minpoint that is located absolutely with respect to the image at the root of the hierarchy above it—that is, the image with the actual memory allocation for the pixels, called the *progenitor*. The reason for this exception is so that absolute coordinates can always be converted to relative coordinates or vice versa. *The important point is that the model uses relative references for subimages by default.*

A more general notion of subimage is also available in the model. It is called a *plyimage* to draw the distinction. It is just like a subimage with the addition of the ability to select a subset of the channels of the parent image. In other words, a plyimage is a subimage in depth as well as height and width. The Altamira COMPOSER method is **image*** PlyImage(**image*** I, **box** b, **channel*** pch). The ply, mask, and permutation of the plyimage are taken from the given **channel** object. Care must be taken to ensure that the permutation given to the plyimage is directly a permutation of the channels of the progenitor image.

[66] The word *cropped* is chosen carefully and is part of the model. The term *clipped* is reserved for geometric modification. So boxes are cropped, but rectangles are clipped. This terminology again is intended to remind us to separate the discrete from the continuous.

IMAGE ALGEBRA

Image or Sprite Assignment

The most fundamental image or sprite operator is *assignment*.[67] I will carefully discuss what it means to assign one image to another in the sprite theory because the concepts involved apply to most other binary image operators and operations.

First, I will give an example of what is *not* meant by image assignment. To *copy* one image to another is not image assignment because no pixels are copied. A copy of an image object is simply a copy of the (programming) object. Thus it is really a special case of a subimage, where the subimage is the entire parent image. Both copies point to the same pixel data. Thus there are two names for the same image. The Altamira COMPOSER method for this is **void** ImageCopy(**image*** Idst, **image*** Isrc).

Image assignment will be denoted in meta-notation by $I = J$ ("*I* gets *J*") for two images *I* and *J*. In general, *I* and *J* have different size, ply, and type. The most general formulation of image assignment consists of three steps: Align, intersect, and assign.

Alignment determines the mapping between array indices of the two images. By default, two images are assumed to have their minpoints aligned. This means that, by default, the upper left pixel of the source (sub)image is mapped to the upper left pixel of the destination (sub)image. Since subimages are images, I will omit the "(sub)" prefix from hereon.

Intersection is the determination of the largest box of pixels shared by the two aligned images. This is computed, of course, by intersecting the two aligned image boxes. Alignment and intersection take into account the fact that two images are generally of different size.

The two box routines referred to previously in the section Special Box Routines, page 44, encapsulate precisely the calculations necessary for these first two steps, hence their importance.

Assignment is the transfer[68] of pixels one-to-one from the source subimage defined by the intersection box to the destination subimage de-

[67] I could say "image or sprite" everywhere but will succinctly use "image" henceforth and assume the reader can supply the "or sprite" versions.

[68] I use "transfer" here to avoid multiple uses of "assignment," but a transfer of one pixel to

fined by that same box. By definition of intersection, there are exactly the same number of pixels in both, and the two subimages have the same size. Assignment must also take into account the fact that in general two images have different ply and type. It must also honor the permutation of the source image and any channel masking that may be in effect.

In practice, alignment and intersection are accomplished with a box algebra computation—one of the two Special Box Routines—preceding the actual transfer with the assignment step. That is, in practice it is convenient to segment the general image assignment problem into two steps, a box algebra step for alignment and intersection, and an image algebra step for actual pixel manipulation. Thus, in the Altamira COMPOSER implementation of the model, the fundamental image assignment method is **image*** Image_J(**image*** I, **image*** J)—also read "*I* gets *J*"—that assigns image *J* to image *I* and returns image *I* or an error (NULL) in case of a problem. This routine assumes alignment and intersection have already been performed and hence, *without checking*, that the given two images are the same size.

In general, the transfer of pixels must take into account that *I* and *J* might have different type and do appropriate conversions during the "transfer." The Altamira COMPOSER implementation makes the simplifying assumption that all images of interest have the same type across an assignment. That is, if a conversion is to be made it can be assumed to have already been made before invoking an assignment (or almost any other binary operation). Thus Image_J( ) further assumes *I* and *J* have the same type.

The problems of different ply and masking across an assignment must be handled. The model has a simple solution for this: The channels of *J* are mapped in order to the channels of *I*, where ordering is determined by the permutation structures of the two images, and masked channels are simply ignored. If *I* has ply greater than *J*, then its excess channels are simply untouched in an assignment from *J*. If *J* has ply greater than *I*, then the assignment simply ceases when there are no more receiving channels in *I*. Once the channels are mapped to one another, then as-

another is just ordinary pixel assignment. That is, a "transfer" leaves the source unchanged.

signment becomes as simple as it sounds: simple transfer of *depth-aligned* channel components from one image to the other.

The generalization to arbitrary binary operator **op** is straightforward. Instead of doing a transfer between the depth-aligned channel components, the operation denoted by **op** is performed. There is typically an assignment of the result of this operation to yet another image, but we already know what it means to do an assignment. The general meta-notation is $I = J \text{ op } K$.

The generalization to n-ary operations, for arbitrary number n of images, is likewise straightforward. In fact, $I = J \text{ op } K$ is an example of a ternary operation. All n images are aligned, intersected, and subimaged to make them the same size. Any conversions are made to make them the same type. Depth alignment is performed and the operation is performed. Typically the last step in the operation is an assignment of the result to a destination image. In the following, several of the most often used operations—those that composite images under the control of yet other images—are explained fully. But first, an example emphasizing box algebra computations is presented.

An Informative Example

With the box algebra machinery and no more image algebra than we have presented so far, we can perform interesting image computations. Here is a good example. See if you can figure out how it works. This is essentially C code directly from Altamira COMPOSER, with certain simplifications to improve readability.

```
image* Image_CycleI(image* I, point p)
{
    /* Cycle given image in place, p.x columns horizontally and
     * p.y rows vertically. p.x, p.y can be negative. The cycle is
     * circular–that is, columns or rows shifted off one side of
     * an image reappear on the opposite side, so no data is discarded.
     */
    box bxoffset, quadbox[4];
    image *psubim, *Quad[4];
```

```
int m;
point c0, c1, Imaxpoint, mask, ptmaxbox, ptoffset, q;
point OnePoint = PointConstruct(1, 1);

if(EqualPoint(p, ZeroPoint))
    return I;

// Make offset modulo the image size
p = ModPoint(p, ImageSize(I));

// Make all shifts positive
p = WherePoint(LTPoint(p, ZeroPoint),
    AddPoint(p, ImageSize(I)), p);
Imaxpoint = MaxPoint(BaseBox(ImageBox(I)));
q = SubPoint(Imaxpoint, p);

for(m = 0; m < 4; m++) {
    // Define four important subimages and save them
    mask = MaskToPoint(m);

    c0 = WherePoint(mask, AddPoint(q, OnePoint), q);
    if(PointToMask(GTPoint(c0, Imaxpoint))) {
        Quad[m] = EmptyImageConstruct( );
        continue;
    }

    c1 = WherePoint(mask, Imaxpoint, ZeroPoint);
    quadbox[m] = BoxFromPoints(c0, c1);

    Quad[m] = ImageConstruct(BoxSize(quadbox[m]),
        ImageChannel(I), IMFLAG_NORMAL);

    psubim = SubImage(I, quadbox[m]);
    Image_J(Quad[m], psubim);
    ImageDestruct(psubim);
}
```

```
        // Restore subimages to new locations
        for(m = 0; m < 4; m++) {
            if(!ValidImage(Quad[m]))
                continue;
            ptoffset = MulPoint(p, MaskToPoint(~m));
            ptmaxbox = AddPoint(ptoffset,
                MaxPoint(BaseBox(ImageBox(Quad[m]))));
            bxoffset = BoxFromPoints(ptoffset, ptmaxbox);
            psubim = SubImage(I, bxoffset);
            Image_J(psubim, Quad[m]);
            ImageDestruct(psubim);
        }

        for(m = 0; m < 4; m++)
            ImageDestruct(Quad[m]);

        return I;
}
```

Image Compositing Operators and "Expressions"

As explained in detail in appendix B, the **over** operator of (Porter & Duff 1984) is fundamental to sprite applications. It is implemented in Altamira COMPOSER by the general routine

image* Image_ImAIpJ(**image*** I, **image*** J, **image*** A)

read as "*I* gets *I* minus *A* times *I* plus *J*." This routine assumes images *I*, *J*, and *A* are the same size and type, as previously explained, and returns image *I*. If *I* and *J* are premultiplied images—that is, sprites—and if *A* is the alpha channel of *J*, then this routine implements $I = J \text{ over } K$. Again, see appendix B for full details, including efficient programming approximations.

The implementation of such a routine must take into account the problems mentioned in the preceding section: *I*, *J*, and *A* may have different ply. *J* or *A* might be an image card (it would not make sense for *I* to be a card).

Although the model does not require it, routines as fundamental as this one should probably be written to be interruptable and to return status information regularly during its execution. These desirable additions are useful so long as an image computation requires seconds or even minutes to execute on an image. This is still true today. Perhaps in a decade, general computing will be so fast as to obviate the need for them. Nearly all Altamira COMPOSER image computation routines are implemented with these features.

Following is a list of sample imaging "expressions" implemented in Altamira COMPOSER. The first two are those already described above. In all cases, the routine returns its first **image*** argument, and the arguments are the necessary images, as **image***s, or other parameters as specified.

Routine Name	Interpretation
Image_J	$I = J$
Image_ImAIpJ	$I = I - A * I + J$
Image_ImAIpAJ	$I = I - A * I + A * J$
Image_ImBIpAJ	$I = I - B * I + A * J$
Image_ImABIpAJ	$I = I - A * B * I + A * J$
Image_ImBIpABJ	$I = I - B * I + A * B * J$
Image_ImABIpABJ	$I = I - A * B * I + A * B * J$
Image_ImCBIpCAJ	$I = I - C * B * I + C * A * J$
Image_JmCBJpCAK	$I = J - C * B * J + C * B * K$
Image_ImCDBIpCDAJ	$I = I - C * D * B * I + C * D * A * J$
Image_AI	$I = I * A$
Image_CDJ	$I = C * D * J$
Image_ImOIpOJ	$I = I - O * I + O * J$, **float** opacity O
Image_ImAOIpOJ	$I = I - A * O * I + O * J$, **float** opacity O
Image_ImAOIpAOJ	$I = I - A * O * I + A * O * J$, **float** opacity O
Image_MaxJK	$I = \max(J, K)$
Image_ImJ	$I = I - J$
Image_IdivA	$I = I/A$
Image_SI	$I = I * S$, **double** scalar S

It is not hard to imagine implementing a language with expressions such as these. Pixar's ICEMAN is such a language, and I had great fun implementing a version in C++ happily overloading the C operators to

implement the languages operators.[69] Altamira COMPOSER does not implement a language. We opted instead to implement the common "expressions" above, plus a handful more, as highly optimized routines and found that this was sufficient for our purposes. In fact, many of these turned out to be superfluous (Smith 1996) in the sense that they were only ever used for a single purpose and were not therefore of general interest.

Image Functions

The "expressions" above are important image functions, but the list of possible functions is infinite. The field of image computation is as large as that of computation. In this section we categorize image functions as an aid to understanding them. The implementations of these ultimately use some of the "expressions" above. The following taxonomy is based on what a function does, rather than how it does it.

Spatial transformations—*transforms*, for short. These are resampling functions (see Continuous Operators on Discrete Sprites, page 29, and Figure 2) that perform geometric spatial operations on a continuous object reconstructed from an image using the Sampling Theorem. These include the familiar scale, rotate, skew, and perspective transforms. Also included is bilinear warping, which maps an image reconstructed into an object with a rectangular bounding box, onto an arbitrary convex quadrilateral before resampling. All of these transforms can be mathematically represented with a 4x4 matrix, and they can be arbitrarily applied in any order. Since there is a small loss of information at each application of a spatial transform (because we cannot practically use the ideal, infinite reconstruction filter required by the Sampling Theorem), it is important not to literally concatenate transforms. Rather, an original source image is maintained; a 4x4 matrix representing the mathematical concatenation of other 4x4 matrices is constructed; the resulting matrix is applied

[69] Not surprisingly, I overloaded ~, &&, and || to implement ¬, ⊓, and ⊔, respectively. As already mentioned, one version of this language I called VAIL, for Volume And Imaging Language.

to the source image so that there is only ever a single generation of loss allowed.

Permutations—*permutes*, for short. The permutes change the order of the pixels within an image but create no new pixels as do resampling functions. There is no loss of information in a permutation. The permutes include flipping an image up-to-down and right-to-left, transposing the pixels of an image about either of its diagonals, rotations (without resampling) clockwise or counterclockwise by exactly 90 degrees, and cyclic shift of the image horizontally or vertically.

Warps. This is a large class of resampling functions. Unlike the spatial transforms, however, repeated application causes deterioration of an image. Included among the warps are barrel and bow distortions, familiar to the video world, and so-called morphing.

Enhancements. These include the classic image processing functions such as brightness and contrast adjustment, tone control, hue and saturation shifts, color balancing, softening, sharpening, and so on. In general, these functions adjust pixel contents without actually changing the pictorial content of an image, the position of its pixels, or its shape.

Textures. These important functions capture the notion of *texturing*[70] one image by the contents of another. So the textures always require two sprites, a source and destination sprite. In the simplest case, color pixels from the source are simply copied to the aligned pixels in the destination. In a slightly more complete implementation, colors *and transparencies* are copied to the destination from the source. In both these cases, the result has the shape essentially of the destination (although in the latter case, the shape can be modified somewhat by transparencies from the source sprite). The shape of the result can only be less than or equal that of the destination. In another variation, the sprites are *glued* together: Clear pixels in the destination *can* be modified by source pixels, within the bounding box of the destination. Another interesting member of this family of functions is *snip*: The shape pixels of the destination sprite

[70] I borrowed this term from 3D computer graphics, where it means that an image is used to give detail to the surface of a 3D geometrical object. Here we use it to mean that a sprite is used to give detail to the "surface" of a(nother) 2D sprite.

are *erased* (cleared, set to 0s) by the overlapping shape pixels of the source. A powerful subclass of texture functions are the *mapping* functions. For example, *transparency map* alters the transparencies of the destination image by the intensities of the color pixels of the source; the destination pixel becomes more transparent where the corresponding source pixel is dimmer. There are many possible variations on this theme. I am surprised how underemployed this powerful class of functions is.

Touchup. This is a large class of functions that can be thought of as simply the hand-driven version of any image computation. For example, so-called "painting" is a touchup function. It is the hand-driven version of a function that simply copies a card to a sprite. If the source is a relatively small image, called a *brush*, and if the position of the brush image relative the destination sprite is determined interactively, say with a mouse or tablet stylus, then we have painting. More accurately, the brush is an alpha image that controls how a card is copied to a destination sprite. I call this class of functions touchup, rather than paint, because the simulation of painting is only one possible hand-driven function. Another is smearing of the image in the direction of brush movement. Another is simple transfer of pixels from one sprite to another under control of the brush as a third, controlling image. If the source pixels always remain constant, then this is called *cloning*. Erasing is the hand-driven version of snipping described above in the textures. There are many possible variations here as well. An extreme is demonstrated in Altamira COMPOSER, where many of the warps can be applied to an image under the handheld brush.

Creation. This class of functions is used to generate new sprites, either from scratch or by deconstructing existing sprites and images. A very popular way of creating sprites is to render them from a 2D or 3D geometrical representation. This automatically generates an accurate and appropriate alpha channel for the resulting sprite. Altamira COMPOSER includes some very simple tools for doing this. These are tools for modeling ellipses, rectangles, splines, and polygons and then rendering them into colored images. I reemphasize that it is the renderings of these 2D

geometrical models that determines shape, not the geometry itself. The geometrical models can be rendered directly into solid color sprites, can be used to snip a given sprite, or can be used to extract pixels from a given sprite by a texturing operation. You can think of this last variation as using the geometrically-derived shape as a "cookie cutter" to extract a new sprite from an older one. Text is another powerful way to create shapes—for example, True Type or PostScript defined text is rendered into shapes that can be used just as the simple geometrically defined ones above are. Another popular method for sprite creation is called *color lifting*.[71] This defines a new sprite shape to be those pixels in a given sprite with colors equal to (or near) a given color. Then the corresponding pixels of the source sprite are "lifted" into the new sprite at those locations.

Rather than continuing to list functions, you can get an idea of the breadth of functionality available by looking at Altamira COMPOSER, of course, but also at any popular image editing application, such as Adobe PHOTOSHOP. Both these applications offer hundreds of functions. In the next section we go the next step beyond single sprites and image editing to multiple sprites and image composition, territory pioneered by Altamira COMPOSER.

Image Composition

In the bad old days, an image required an expensive piece of memory for storage. So an image connoted large and expensive. If there were enough memory to hold an image, then a user did things to it, passed filters over it, painted on it, etc. I call this the *monolithic* model of image computing—an image as a single large, heavy stone. Most imaging applications in the market today are still built on the monolithic conception—Adobe PHOTOSHOP being the best known. This is to be contrasted with the new mental model that is now appropriate. I could coin a term and call it the *polylithic* model but I think the idea is better carried by simply thinking of a stack of brightly colored pebbles—small, light, and numerous—that can be rear-

[71] Often called, for no meaningful reason, "magic wand." I will use a meaningful term.

ranged at will. Then, *in addition to* the image editing functions of yore (the monolithic era) are all the functions needed for arrangement of mutiple images—sprites as we are calling these nimble things. Thus image editing graduates into *image composition*. Thus an image composition application can be thought of as a tool for creating arrangements, or compositions, of sprites, with a full image editor available for each sprite. You will not be surprised to learn that Altamira COMPOSER was the first such application.

So in this section, I extend our model to include compositions of sprites and functions for working with compositions. I will borrow heavily from an existing class of picture composition tools in the computer software market for our model. This is the class of geometry-based applications called drawing, or illustration, programs. In a drawing program, a user works in a creative space (see Creative Space vs Display Space, page 15) that is 2D and infinite in all directions. Objects are placed in this space arbitrarily. For example, a triangle here, a cylinder there, and a sphere between them. These objects are, of course, geometrical objects, not sprites.

Our model is exactly the same as that for the drawing programs but with sprites (image objects) substituted for geometry objects. This very simple, picture-based idea has been passed up for years in the imaging world, which still insists on using a much less appropriate text-based metaphor that assumes everything is pasted down into a single (monolithic) object and unpasted only temporarily when "selected." Old selections are lost and have to be re-extracted for future use. Adobe PHOTOSHOP is the classic example of this paradigm (although introduction of "layers" alleviated this restriction somewhat).

Let's look further at the drawing metaphor that we shall adopt in our model. There is nothing behind the geometrical objects in the creative space. In a display of the creative space, something has to be displayed behind them. It is the so-called "desktop." It is just whatever is back there. When printed to paper, the white paper shows there. This is called the *abyss* in our model.[72] The abyss, or desktop, is not printed. It is not part of the creation. One unfortunate legacy of the monolithic era is

[72] I have also called this the *void*, which I like as an alternative.

the notion that images are always rectilinear and that, therefore, there is always a background image "back there." In the sprite theory model, there is no need for a background image. One can always designate a given sprite or sprites to be the "background" but this is only convenient sometimes for naming and not part of the model. When printing a composition of sprites, the abyss conceptually doesn't print.[73] Just as the geometrical drawing applications have always not printed the desktop.

Another unfortunate legacy of the monolithic era is the notion that images have edges. In particular, the edges of the rectilinear background image define the extent of a "composition." If a "selection" is dragged past the edge of the background, it is automatically cropped to that edge.[74] In our model, there are no edges to a composition in creative space. Any edges are an artifact of the decision about which part of a composition is to be displayed in display space. It is still usually true that displays (monitors, film frames, video frames, photographs, pieces of paper) tend to be rectilinear. The point is that this restriction does not have to be invoked until a display space decision is made. Again, this is exactly how creation and display are divided from one another in geometry applications. In summary, in our model, cropping and pasting only happen at the last step, and only when instructed to do so by the user/creator.

Functions that we borrow directly from the geometrical forebears are depth ordering, alignment, and multiple selection. See any geometry program for how these work. By the way, "selection" becomes, in this model, simply pointing to a sprite—say, by clicking on it—just as in the geometry programs. Since objects are not pasted together, they are always available with a simple click.

There are two notions of sets of sprites in the model: There are *multiple selections* of sprites, and there are *groups* of sprites. A multiple selection is a set of sprites that are treated as individuals. Thus a rotate-about-

[73] Of course, real printers don't composite their inks with the paper so the print function has to accommodate the color or image of the page being printed on. But this is an output or display problem, separated conceptually from the creative step.

[74] This is the acid test of an application to see if it has graduated out of the Monolithic Age. Try it on your favorite imaging app and see what happens.

center function applied to a multiple selection causes each sprite in the set to rotate individually about its center. A group, on the other hand, is a set of sprites treated as if they constituted a single sprite. The same rotate function applied to a group would rotate the entire set around the center of the single "meta-sprite." There can be groups of groups but not multiple selections of multiple selections. That is, a group can have hierarchical structure, but a multiple selection cannot.

Image Composition Display

The sprite model celebrates the notion that the creation of a composition and the display of it are two separate processes (cf., page 15). Many of the features outlined above for image composition are performed by a designer interacting with the display of the composition. The sprites themselves reside in unknown and arbitrary locations in a computer or network. The actual pasting of the sprites together to form a final composite is the last process performed by the designer when the design is known to be complete. It is called *flattening*. But the *display* of a composition appears to paste them together at all times. It is this fiction that makes the app work, because it appears to the user that all the sprites are actually in one space, as designed.

Altamira COMPOSER borrows yet another notion from the geometrical modeling world. It lets the user have multiple views of a composition. Notice that this is just like having multiple cameras looking at a 3D synthetic scene in classic computer graphics. Of course, this is just a restatement of the creative vs display space notion. The important point is that each view, or display, must be recomputed every time a change is made to the composition. A recomposite of all visible sprites must occur for each view. Just keep in mind that this is for display only. The official flattening happens only under user directive, presumably as a last step, because it is difficult if not impossible to reverse (and why we argue against the old text-based paradigm that forces one to do exactly that).

So the rapid composition and recomposition of sprites is fundamental to a pleasing display of an image composition application. In our model, *the current sprite* is the one that an indicated operation happens to

(or current sprites in case of a multiple selection). It can be at any level in the depth ordering of the composition. A good composition algorithm must take this into account. A tour de force example of the use of box algebra and the over image composition operator is presented in detail in (Smith 1990). I will not present the details here. Suffice it to say that the algorithm presented there seeks to minimize the number of pixels that actually have to be touched to do a recomposite, and this requires taking careful note of depth information.

Sprite Picking

One of the problems I had to solve that may not seem to be a problem is that of picking a particular sprite in a display of sometimes dozens or even hundreds or thousands of partially overlapping, partially transparent sprites. How does one do it? I tried several different schemes before hitting on the one described in detail in (Smith 1990) and adopted into our model.

The basic notion is that when there is no ambiguity, then a click within the bounding box of a sprite is sufficient to select, or pick, it. This is true even if the sprite is mostly clear and you click on the abyss. Thus you can be sloppy in your picking if there is no ambiguity. If you click on the abyss and are not within the bounding box of any sprite, then you simply *miss*.

In the case of ambiguity, the topmost pixel with non-0 alpha that you touch picks the corresponding sprite. Thus it is the shape of a sprite that is used to determine if picking has occurred. A way to say this is: *If you can see it, you can pick it*—so long as you understand that seeing through a partially transparent object does not count. The partially transparent object would be picked instead.

If you click on a stack of sprites, but on an abyss pixel showing through them all, then it is the bottom sprite you get, assuming the click is inside its bounding box.

This picking scheme has worked very well, so well—so intuitively—that it is a surprise perhaps to discover that it had to be invented, and that imaging applications don't use it.

Future Directions

I indicated earlier that once we clearly understand 2D imaging then we can proceed to integrate it with 2D geometry. I would like to discuss this again now that we have the full machinery of our model at hand. The important point is that the old monolithic model did not lend itself to an integration with geometry, but the polylithic, sprite theory model does. The careful separation of geometry and sampling makes the convergence easier because we know exactly what we are doing.

So the future looks like this: 2D geometric objects are added to compositions of 2D sprites (sampled objects). Since they are both objects and both displayed in image space, this is straightforward. What has to be added are definitions of interactions between such objects. For example, what does it mean to paint on a 2D geometrical object? What does it mean to glue a geometric object to an image object? What does it mean to blur a geometric object. And so forth. I believe that the answers to these questions are generally straightforward, once we have the common space in which to speak of them and are careful of the distinctions between them.

Then why not add 3D geometry (or even 3D sampling) objects? Sure. And then sound? Again, sure. And animation and interaction? Sure. In the common space advocated here the digital convergence is easy to visualize.

APPENDIXES

A: A PIXEL IS *NOT* A LITTLE SQUARE, A PIXEL IS *NOT* A LITTLE SQUARE, A PIXEL IS *NOT* A LITTLE SQUARE! (AND A VOXEL IS *NOT* A LITTLE CUBE)[75]

ABSTRACT

My purpose here is to, once and for all, rid the world of the misconception that a pixel is a little geometric square. This is not a religious issue. This is an issue that strikes right at the root of correct image (sprite) computing and the ability to correctly integrate (converge) the discrete and the continuous. The little square model is simply incorrect. It harms. It gets in the way. If you find yourself thinking that a pixel is a little square, please read this paper. I will have succeeded if you at least understand that you are using the model and why it is permissible in your case to do so (is it?).

Everything I say about little squares and pixels in the 2D case applies equally well to little cubes and voxels in 3D. The generalization is straightforward, so I won't mention it from hereon.[76]

I discuss why the *little square model* continues to dominate our collective minds. I show why it is wrong in general. I show when it is appropriate to use a little square in the context of a pixel. I propose a discrete to continuous mapping—because this is where the problem arises—that always works and does not assume too much.

I presented some of this argument in (Smith 1995b) but have encountered a serious enough misuse of the little square model since I wrote that paper to make me believe a full frontal attack is necessary.

[75] This is (Smith 1995c), originally written July 17, 1995.

[76] I added the "voxel" phrase to the title on Nov. 11, 1996, after attending the Visible Human Project Conference 96 in Bethesda, Md, where I heard "voxel" misused as a little cube many times.

A: A PIXEL IS *NOT* A LITTLE SQUARE!

THE LITTLE SQUARE MODEL

The little square model pretends to represent a pixel (picture element) as a geometric square.[77] Thus pixel (i,j) is assumed to correspond to the area of the plane bounded by the square $\{(x,y)|i-.5 \le x \le i+.5, j-.5 \le y \le j+.5\}$.

I have already, with this simple definition, entered the territory of controversy—a misguided (or at least irrelevant) controversy as I will attempt to show. There is typically an argument about whether the pixel "center" lies on the integers or the half-integers. The "half-integerists" would have pixel (i,j) correspond instead to the area of the plane $\{(x,y)|i \le x \le i+1., j \le y \le j+1.\}$.

This model is hidden sometimes under terminology such as the following—the case that prompted this memo, in fact: The resolution-independent coordinate system for an image is $\{(x,y)|0. \le x \le W/H, 0. \le y \le 1.\}$, where W and H are the width and height of the image. The resolution dependent coordinate system places the edges of the pixels on the integers, their centers on the edges plus one half, the upper left corner on $(0.,0.)$, the upper right on $(W,0.)$, and the lower left on $(0.,H)$. See the little squares? They would have edges and centers by this formulation.

SO WHAT *IS* A PIXEL?

A pixel is a *point* sample. It exists only at a point. For a color picture, a pixel might actually contain three samples, one for each primary color contributing to the picture at the sampling point. We can still think of this as a point sample of a color. But we cannot think of a pixel as a square—or anything other than a point. There are cases where the *contributions* to a pixel can be modeled, in a low-order way, by a little square, but not ever the pixel itself.

An image is a rectilinear array of point samples (pixels). The marvelous Sampling Theorem tells us that we can reconstruct a continuous ent-

[77] In general, a little rectangle, but I will normalize to the little square here. The little rectangle model is the same mistake.

ity from such a discrete entity using an appropriate *reconstruction filter.*[78] Figure 2 illustrates how an image is reconstructed with a reconstruction filter into a continuous entity. The filter used here could be, for example, a truncated Gaussian. To simplify this image, I use only the *footprint* of the filter and of the reconstructed picture. The footprint is the area under the non-0 parts of the filter or picture. It is often convenient to draw the minimal enclosing rectangle for footprints. They are simply easier to draw than the footprint—Figure 2(d). I have drawn the minimal rectangles as dotted rectangles in Figure 2.

Figure 3 is the same image reconstructed with a better reconstruction filter—for example, a cubic filter or a windowed sinc function—and not an unusual one at all. Most quality imaging uses filters of this variety. The important point is that both of these figures illustrate valid image computations. In neither case is the footprint rectangular. In neither case is the pixel ever approximated by a little square. If a shape *were* to be associated with a pixel (and I am not arguing that it should), then the most natural thing would be the shape of the footprint of the reconstruction filter. As these two examples show, the filters typically overlap a great deal.

Figure 4 is the same image reconstructed with one of the poorest reconstruction filters—a box filter. The only thing poorer is no reconstruction at all—resulting in the abominable "jaggies" of the early days of computer graphics—and we will not even further consider this possibility. The Figure 4 reconstruction too is a valid image computation, even though it is lacking in quality. This lowest-quality case is the only one that suggests the little square model.

So it should be clear that the coordinate system definition given above is not suitable for anything but the lowest-quality image computing. The edges of a reconstructed image are most naturally defined to be its minimally enclosing rectangle. But these are dependent on the chosen reconstruction filter.

The only resolution independent coordinate system that one can safely map to an image requires a known reconstruction filter. Given an im-

[78] And some assumptions about smoothness that we do not need to worry about here.

age, say that represented by Figure 3(a), and a known filter, say that of Figure 3(b), then I can determine exactly what the minimally enclosing rectangle is and map this to the normalized rectangle of the proposed definition above: $\{(x,y)|0. \le x \le W/H, 0. \le y \le 1.\}$. Then the pixel (point sample, remember) locations can be backed out of the mapping. Will they sit on the half-integers? In the three cases above, all of which are valid, only Figure 4 (the worst) will have the samples on the half-integers under this mapping. Will the left edge of the reconstructed image lie distance .5 left of the leftmost column of pixels? Again, only in the worst case, Figure 4.

I would suggest at this point that the only thing that is fixed, in general, are the samples. Doesn't it make sense that they be mapped to the integers since that is so simple to do? Then the edges of the reconstructed continuum float depending on the chosen filter. I believe that if you rid yourself of the little square model, it does not even occur to you to put the samples on the half-integers, an awkward position for all but the lowly box filter. The bicubic filter that I most often use has a footprint like the minimal enclosing rectangle of Figure 3(b). Half-integer locations for this filter are simply awkward. Certainly one can do it, but why the extra work?

I believe that the half-integer locations are attractive for "little-squarists" because the minpoint (upper left corner) of the reconstructed entity falls at $(0,0)$. But note that this only happens for—yes, again—the box filter. For my favorite filter, the minpoint would fall at $(-1.5,-1.5)$. Is that more convenient, prettier, or faster than $(-2.,-2.)$? No.

WHY IS THE LITTLE SQUARE MODEL SO PERSISTENT?

I believe there are two principal reasons that the little square model hasn't simply gone away:

Geometry-based computer graphics uses it.

Video magnification of computer displays appears to show it.

Geometry-based computer graphics (3D synthesis, CGI, etc.) has solved some very difficult problems over the last two decades by assum-

ing that the world they model could be divided into little squares. *Rendering* is the process of converting abstract geometry into viewable pixels that can be displayed on a computer screen or written to film or video for display. A modern computer graphics model can have millions of polygons contributing to a single image. How are all these millions of geometric things to be resolved into a regular array of pixels for display? Answer: Simplify the problem by assuming the rectangular viewport on the model is divided regularly into little squares, one per final pixel. Solve the often-intense hidden surface problem presented by this little square part of the viewport. Average the results into a color sample. This is, of course, exactly box filtering. And it works, even though it is low order filtering. We probably wouldn't be where we are today in computer graphics without this simplifying assumption. *But*, this is no reason to *identify* the model of geometric contributions to a pixel with the pixel. I often meet extremely intelligent and accomplished geometry-based computer graphicists who have leapt to the *identification* of the little square simplification with the pixel. This is not a plea for them to desist from use of the little square model. It is a plea for them to be aware of the simplification involved and to understand that the other half of computer picturing—the half that uses no geometry at all, the imaging half—tries to avoid this very simplification for quality reasons.

When one "magnifies" or "zooms in on" an image in most popular applications, each pixel appears to be a little square. The higher the magnification or the closer in the zoom, the bigger the little squares get. Since I am apparently magnifying the pixel, it must be a little square, right? No, this is a false perception. What is happening when you zoom in is this: Each point sample is being replicated $M \times M$ times, for magnification factor M. When you look at an image consisting of $M \times M$ pixels all of the same color, guess what you see: A square of that solid color! It is not an accurate picture of the pixel below. It is a bunch of pixels approximating what you would see if a reconstruction with a box filter were performed. To do a true zoom requires a resampling operation and is much slower than a video card can comfortably support in realtime to-

day.[79] So the plea here is to please disregard the squareness of zoomed in "pixels." You are really seeing an $M \times M$ array of point samples, not a single point sample rendered large.

HOW DOES A SCANNER DIGITIZE A PICTURE?

Just to be sure that I eradicate the notion of little square everywhere, let's look at the scanning process. I want to be sure that nobody thinks that scanners work with little squares and that, therefore, it is all right to use the model.

A scanner works like this: A light source illuminates a piece of paper containing a colored picture. Light reflected from the paper is collected and measured by color sensitive devices. Is the picture on the paper divided up into little squares, each of which is measured and converted to a single pixel? Not at all. In fact, this would be very hard to accomplish physically. What happens instead is that the illuminating light source has a shape or the receiving device has an aperture that gives the incoming light a shape or both, and the device integrates across this shape. The shape is never a square. It is not necessarily accurate, but will give you the correct flavor, to think of the shape as a Gaussian. In general, overlapping shapes are averaged to get neighboring pixel samples.

So scanning should not contribute any weight to the little square model.

HOW DOES A PRINTER PRINT A DIGITAL IMAGE?

Again, let's look at a familiar process to determine if it contributes to the little square model. The process this time is printing. This is a very large and complex subject, because there are many different ways to print an image to a medium.

Consider printing to film first. There are several ways to do this. In any process where a flying spot of light is used to expose the film, then

[79] This remark is dated now. Realtime magnification is now readily available but simple pixel replication is still used remarkably often for "magnification."

the exposing beam has a shape—think of it as Gaussian. There are no little squares in this process.

Consider half-tone printing of ink on paper. The concept here is to convert a pixel with many values to a dot of opaque ink on paper such that the area of the dot relative to a little square of paper occupied by the dot is in the ratio of the intensity of the pixel to the maximum possible intensity. Sounds like the little square model, doesn't it? But for color printing, the different primaries are printed at an angle to each other so that they can "show through" one another. Therefore, although there are little squares in each color separation, there are none for the final result.

There is a new technique for printing ink-on-paper that uses stochastic patterns within each little square. Hence the separations do not have to be rotated relative one another in order to "show through." The little square model is a decent model in this case.

There are sublimation dye printers and relatives now that print "continuous tone" images in ink on paper. I believe that these use little squares of transparent dyes to achieve continuous tone. In that case, they probably use the little square model.

The point here is that the use of the little square is a printing technology decision, not something inherent in the model of the image being imaged. In fact, the image being imaged is simply an array of point samples in all cases.

HOW IS AN IMAGE DISPLAYED ON A MONITOR?

Many people are aware that a color monitor often has little triads of dots that cause the perception of color at normal viewing distances. This is often true, except for Sony Trinitrons that use little strips of rectangles rather than triads of dots. In neither case are there little squares on the display. I will assume triads for this discussion. It goes through for the Trinitron pattern too, however.

While we are at it, I would also like to dispel any notion that the triads are pixels. There is no fixed mapping between triads and pixels driving them. The easiest way to understand this is to consider your own graphics card. Most modern cards support a variety of different color

resolutions—for example, 640×480, 800×600, 1024×768, etc. The number of triads on your display screen do not change as you change the number of pixels driving them.

Now back to the display of pixels on a screen. Here's roughly what happens.[80] The value of a pixel is converted, for each primary color, to a voltage level. This stepped voltage is passed through electronics which, by its very nature, rounds off the edges of the level steps. The shaped voltage modulates an electron beam that is being deflected in raster fashion across the face of your display. This beam has shape—again think of it as Gaussian (although it can get highly distorted toward the edges of the display). The shaped beam passes through a shadow mask that ensures that only the red gun will illuminate the red phosphors and so forth. Then the appropriate phosphors are excited and they emit patterns of light. Your eye then integrates the light pattern from a group of triads into a color. This is a complex process that I have presented only sketchily. But I think it is obvious that there are no little squares involved at any step of the process. There are, as usual in imaging, overlapping shapes that serve as natural reconstruction filters.

Another display issue that implies the little square model is the notion of displays with "non-square pixels." Although it is becoming less common now, it used to be fairly common to have a video display driven by, say, a 512×480 image memory. This means that 512 samples are used to write a row to the display, and there are 480 rows. Video monitors have, if correctly adjusted, a 4:3 aspect ratio.[81] So the *pixel spacing ratio* (*PSR*) for this case is

$$\frac{\left(\frac{4}{512}\right)}{\left(\frac{3}{480}\right)} = \left(\frac{4}{3}\right) * \left(\frac{480}{512}\right) = 1.25.$$[82]

[80] This paragraph appears dated now, referring to the almost universal use of cathode ray tubes (CRTs) for display. But most color display technologies have some related broadening characteristic.

[81] Aspect ratio is the ratio of display width to height.

[82] If you look closely at this computation you will discover yet another application of the pernicious "little square" model—actually a "little rectangle" model. It assumes that 512 "pixels" are mapped to 4 units and 480 of them to 3 units. In other words, it assumes a (problematical) mapping of "pixels" to a geometric rectangle. Nevertheless, 1.25 is a decent approximation of PSR in this case.

So the correct terminology for this case is that the monitor has a "non-square pixel spacing ratio," not that it has "non-square pixels." Most modern computer displays, if correctly adjusted, have square PSR—that is, PSR = 1.

So we have no contributions from scanning or display processes for the little square model. We have a case or two for particular printing technologies that support a little square model, but they are not the general printing case. In summary, the processes used for image input and output are not sources for the little square model.

WHAT IS A DISCRETE TO CONTINUOUS MAPPING THAT WORKS?

The only mapping that I have been able to come up with that doesn't assume too much is this: Assume the samples are mapped to the integers (it's so easy after all—just an offset to the array indices used to address them in an image computation). Then the outer extremes of the image are bounded by a rectangle whose left edge is $(filterwidth/2)$ to the left of the leftmost column of integers, right edge is $(filterwidth/2)$ to the right of the rightmost column of integers, top edge is $(filterheight/2)$ above the topmost row of integers, and bottom edge is $(filterheight/2)$ below the bottommost row of integers occupied by image samples).

This is still not quite complete, because it assumes symmetric filters. There are cases—for example, perspective transformations on (reconstructions of) images—when asymmetric filters are useful. So the mapping must be generalized to handle them. By the way, by symmetrical filter I mean that it is symmetrical about its horizontal axis, and it is symmetrical about its vertical axis. I do not mean that it is necessarily the same in both dimensions.

I do believe that it is important to have the vertical dimension increase downward. It is not required, but it is so natural that I find it hard to argue against. We display from top to bottom on millions of TVs and computer displays. We read from top to bottom. We compute on matrices—the natural container for images—from top to bottom. Nearly all

popular image file formats store images from top to bottom,[83] or if in tiled form, in rows of tiles from top to bottom.

IMAGE BROADENING

I mentioned earlier that I would return to a discussion of whether the minimal enclosing rectangle of a reconstructed image were the most natural representation of it. This will fall out of a discussion of image *broadening*.

By the very nature of the Sampling Theorem that underlies everything that we do, during a reconstruction an image suffers broadening due to the width of the reconstruction filter. Again, look at Figure 2–Figure 4 for examples. The amount of broadening is dependent on the filter used. If the filter is asymmetric, then so is the broadening.

I call the excess of a broadened image over itself to be its *margin*. In general then, an image margin is not symmetric about its image and is dependent on the reconstruction filter being used.

Let's be concrete: Consider scaling 640×480 image down by a factor of 2 in both dimensions. The original image is assumed to have its minpoint at (0, 0) and its maxpoint (lower right corner) at (639, 479). Assume we are using a symmetric bicubic filter for good results. Its canonical footprint is $\{(x,y)|-2. \leq x \leq 2., -2. \leq y \leq 2.\}$. Then to do the scaling we ***reconstruct-transform-resample***:[84] When we reconstruct our image into a continuum, it occupies the space from –2. to 641. horizontally and from –2. to 481. vertically. Thus its minimal enclosing rectangle has minpoint (–2., –2.) and maxpoint (641., 481.). If we were to resample at this moment, before doing any transform on the reconstruction, we would have broadened the image by one pixel all around; we would have introduced a 1-pixel margin by the act of reconstruction. (The filter has 0 weight at its extremes so the samples along the minimal enclosing rectangle would be 0 and not figure into the support of the sampled image.)

[83] The Targa file format (.tga) is the most flagrant violator of this rule; it also reverses RGB to BGR. And WINDOWS has adapted the Targa format for its "bitmaps" (without documenting the color channel reversal, as I unhappily discovered as a developer).

[84] Generally, there needs to be a low-pass filtering step too, before resampling. When scaling up, no high frequencies are introduced, but when scaling down, they are and must be low-pass filtered to satisfy the Sampling Theorem.

This is a good point to pause and note that it is not necessarily the broadened footprint that is interesting. In this case, if we have only a 640×480 display then we would crop off the margin pixels for redisplay anyway. In my experience, I usually want to know about an image's extent and its margins, before and after transformations, in full detail so that I can decide exactly what I want to do. The important point is that this information be available, not how.

Back to the scaling example: Now let's minify the reconstruction by 2. This means that the reconstruction is scaled down about its center. This is easily modeled by scaling down the minimal enclosing rectangle that I will represent as (–2., –2.) → (641., 481.). The scaling happens about the center at (319.5, 239.5). There are many ways to proceed from here so I will not pursue the details. I believe that I have presented enough of the problem and technique of solution that it is clear that nothing is offered to it by a little square model for the pixels.

In fact, the little square model might have misled us into a compounding of low quality techniques: It is tempting to box filter box filters. Thus it is tempting to take the 640×480 example above and look at each set of 2×2 pixels to scale down by 2. It is another application of the box filter to simply average each 2×2 set of pixels into a single new pixel. This is generally a bad idea and not the path to take for good results.

SUMMARY

I have presented a brief but inclusive analysis of sampling and filtering. It has been shown that the little square model does not arise naturally in sampling theory, the main underpinning of everything we do. It does not arise in scanning or display. It arises in printing only in restricted cases. The geometry world uses it a great deal because they have had to simplify in order to accomplish. Their simplified model of *contributions* to a pixel should not be confused with or identified with the pixel. Magnified screen pixels that look like little squares have been shown to be a quick and dirty trick (pixel replication) by graphics boards designers, but not the truth. In short, the little square model should be suspect whenever it is encoun-

tered. It should be used with great care, if at all, and certainly not offered to the world as "standard" in image computing.

In the process of this presentation, I have demonstrated at least one very natural mapping of the discrete samples of an image to a continuum reconstructed from it. The method used is straightforward and always works. I have also argued that the following details of such a reconstruction are interesting enough to track: the exact filter used, the image margins created by reconstruction, and minimal enclosing rectangles. I have also suggested that it is natural for the vertical coordinate to increase downwards, and that it is simple and natural for sample locations to be mapped to the integers. However, neither of these is required, so long as consistency reigns. I do argue, however, that placing samples at the half-integers seems to indicate a lurking reluctance to dismiss the little square model, or the box filter, the only two common situations where the half-integers are natural concepts.

Finally, I have pointed out two related misconceptions: (1) The triads on a display screen are not pixels; they do not even map one-to-one to pixels. (2) People who refer to displays with non-square pixels should refer instead to non-square, or non-uniform, pixel spacing.

B: IMAGE COMPOSITING FUNDAMENTALS[85]

ABSTRACT

This is a short introduction to the efficient calculation of image compositions. Some of the techniques shown here are not well known, and should be. In particular, we will explain the difference between premultiplied alpha and not.[86] These two related notions are often confused, or not even understood. We shall show that premultiplied alpha is more efficient, yields more elegant formulas, and occurs commonly in practice. We shall show that the non-premultiplied alpha formulation is not closed on **over**, the fundamental image compositing operator—as usually defined. Most importantly, the notion of premultiplied alpha leads directly to the notion of *image object*, or *sprite*—a shaped image with partial transparencies.

THE BASIC MODEL

There are two ways to think of the alpha of a pixel. As is usual in computer graphics, one interpretation comes from the geometry half of the world and the other from the imaging half. Geometers think of "pixels" as geometrical areas intersected by geometrical objects.[87] For them, alpha is the percentage *coverage* of a pixel by a geometrical object. Imagers think of pixels as point samples of a continuum. For them, alpha is the *opacity* at each sample. In the end, it is the imaging model that dominates, because a geometric picture must be reduced to point samples to display—it must be rendered. Thus, during rendering coverage is always converted to opacity, and all geometry is lost.

[85] The early parts of this appendix are redundant with the main part of this lecture, but I choose to ignore the redundancy to keep the appendix true to the original paper (Smith 1995a). To avoid most of the redundancy, go immediately to the section Composite Alpha.

[86] These are called *associated* and *unassociated* alpha as well. I can never remember which is which so don't use the terms.

[87] A little square is a very common model for the "pixel." I place this term in quotes to remind us that this is not a pixel (a sample) but a model for possible geometric contributions to the final sample. The last thing I want to promulgate is the notion that a pixel is a little square.

B: IMAGE COMPOSITING FUNDAMENTALS

The Porter-Duff matting algebra (Porter & Duff 1984) that underlies what we present here is based on a model that is easiest to understand by alternating between the two conceptions.[88]

The elementary imaging operation that we wish to elaborate is called, in (Porter & Duff 1984), the **over** operator. It captures the notion of compositing image J over image I, where either I or J or both may be partially transparent. For ease, we will think of images I and J as being rectilinear, the same size, and each having four channels—three for RGB color and one for alpha (that is, opacity).

Think of the following geometrical model: A "pixel" is an area α percent covered by an opaque geometrical object with color A. Thus the amount of color contributed by that area is αA. That is, we average the color over the "pixel" and come up with a single new color representing the entire area—the color αA is a point sample.

Now think of another opaque geometrical object with color B added to the original "pixel" area. Disregard for a moment the other geometrical object there. Assume that the new geometrical object has coverage of the "pixel" equal to β. So the "pixel" is contributing color βB due to this object. This again is a point sample representing the color of the second object.

But now we use the geometry model to conceptually combine the contributions of the two objects in the "pixel" area. The second object is allowing only $(1-\beta)$ percent of the "pixel" area to be transparent to any objects behind it. We simply ignore the actual geometry of the two objects at this point and assume that, in general, the "pixel" is allowing $(1-\beta)$ times the color from behind, αA, to show. This is added to the color due to the top object βB. So the total color of object with color B over object with color A is $\beta B + (1-\beta)\alpha A$.

Notice that this result could be completely wrong if the geometry of the second object exactly coincided with that of the first. The bottom color would not contribute at all to the final color in this special case. So

[88] The Porter-Duff paper is an excellent example of why the little square model for contributions to a pixel has become confused, in the geometry-based computer graphics world, with the pixel itself. All illustrations in that paper use the little square model. A unit circle could have been used equally effectively, however—or any other unit area region.

the model we are using is an approximation for the general case of combining two images where we no longer have any idea of how the alpha at a point was determined. In an image there is no way to tell whether a point sample with a partial opacity comes from a partially transparent surface or from an opaque surface partially occluding the area represented by the point sample.

PREMULTIPLIED ALPHA

The formula we have just derived from basic principles is this: For composite color C obtained by placing a pixel with color B and alpha β over a pixel with color A and alpha α:

$$C = \beta B + (1-\beta)\alpha A = \beta B + \alpha A - \beta\alpha A.$$

Notice how many multiplies this formula implies—three[89] at each pixel for each color component. Considering that this formula is extremely basic to computer graphics and that multiplies are expensive,[90] the early researchers at Lucasfilm and Pixar observed that this formula could be reduced to one multiply per pixel per component if the alphas were *premultiplied* times the color of an image. That is, if the color channels of image I contained, not color A, but weighted color αA, and similarly for image J, then the formula above reduces to

$$C' = B' + (1-\beta)A' = B' + A' - \beta A'$$

where the primes indicate colors have been premultiplied by their corresponding alphas. The images are said to have *premultiplied alphas*. Of course, it is the color channels that are different, not the alpha channels, despite this terminology.

There is a subtlety here that will cause trouble if not identified. We have called the resulting color here C' as if it were different from the color C computed above in the non-premultiplied alpha case. *But it isn't!* It is the same computation, where entities on the right have been abbreviated because of premultiplication. We will return to this problem later.

---

[89] Two, actually, with a little rearrangement: $T = \alpha A, C = \beta(B-T) + T$.

[90] They were especially expensive then. Now we would just like to avoid extra steps.

B: IMAGE COMPOSITING FUNDAMENTALS

Images with premultiplied alphas have been used for many years very successfully by Lucasfilm, Pixar, and Altamira in hundreds of thousands, if not millions, of images. The TIFF image storage format is aware, as of version 6.0, of premultiplied alphas.

COMPOSITE ALPHA

We have given the formulas above for the color channels in a composite of two partially transparent images. What is the composite alpha channel formula? Notice that it will be the same for both cases, since premultiplication only applies to the color channels.

The same model as used above for composite color can be used for composite alpha. The average opacity of the "pixel" partially covered by the first geometric object is β, and that for the second geometric object is α. But the geometry of the model allows only $(1-\beta)$ of the lower light filter to be effective. So the composite alpha is

$$\gamma = \beta + (1-\beta)\alpha = \beta + \alpha - \alpha\beta$$

in either case, premultiplied or not.

AN ELEGANT FORMULATION AND A FLAWED ONE

Let's collect together the results from above.

Compositing Formulas for **over**, Colors Not Premultiplied by Alpha:

$$C' = \beta B + (1-\beta)\alpha A = \beta B + \alpha A - \beta\alpha A$$
$$\gamma = \beta + (1-\beta)\alpha = \beta + \alpha - \alpha\beta$$

Compositing Formula for **over**, Colors Premultiplied by Alpha:

$$C' = B' + (1-\beta)A' = B' + A' - \beta A'$$

In the latter case, we need only one formula to represent the color channels *and* the alpha channel, a more elegant formulation certainly than the former case that requires a formula for the color channels different from that for the alpha channel.

Now we will see why the former case is flawed.

THE "SECOND-COMPOSITION" PROBLEM

You may have noticed that this time I used C' for the left side of the non-premultipled colors case, since it has already been observed that C and C' are the same color in either formulation. Recall that the prime indicates a color that has been premultiplied by its alpha. So you see the problem: The first formulation maps non-premultiplied colors into premultiplied colors. The second maps premultiplied colors into premultiplied colors. In other words, the usual definition of **over** for non-premultiplied images is not closed on **over**, a problem we will fix below.

This problem is called the "second-composition" problem because it shows up in second (or subsequent) compositions using results from first (or earlier) compositions. Let image K be the result obtained above for J **over** I. Suppose we want to perform a second non-premultiplied composition of L **over** K, where image L has color D and alpha δ. In order to use the formula above we need the non-premultiplied color of K and its alpha δ. The non-premultipled color of K is C' divided by δ. So the *second-composition* formula is

2nd-Compositing Formulas for **over**, Colors Not Premultiplied by Alpha:

$$E' = \delta D + (1-\delta)\gamma(C'/\gamma) = \delta D + (1-\delta)C' = \delta D + C' - \delta C'$$

The alpha channel calculation is as before, and the premultiplied case works as before. That is, there is no second-composition problem for the premultiplied case—another example of its relative elegance.

But the non-premultiplied case is a mess. One either has to divide through by the new alpha at each pixel in order to use the original (first-composition) formulas, or one has to carry around a mixture of premultiplied and nonpremultiplied information to use the simpler second-composition formulas.

So here are the cleanest formulations for the two cases, where we do not clutter up our minds with two different models during the course of a series of compositions and where there is no need for the confusing

first- and second-composition distinction—that is, these formulations are closed on **over**:[91]

Closed Compositing Formulas for **over**, Colors Not Premultiplied by Alpha:

$$\gamma = \beta + (1-\beta)\alpha = \beta + \alpha - \alpha\beta$$
$$C' = \beta B + (1-\beta)\alpha A = \beta B + \alpha A - \beta\alpha A$$
$$C = C'/\gamma$$

Closed Compositing Formula for **over**, Colors Premultiplied by Alpha:

$$C' = B' + (1-\beta)A' = B' + A' - \beta A'$$

Many practitioners are unaware of the second-composition problem because they often only do one composition—for example, as the last stage of a 3D rendering project: all the objects are rendered as sprites,[92] then they are composited, and never used again. Or more importantly, composites of them are never used for future composites. It is the modern world of cheap memory that has made it possible and common to recomposite a set of sprites many times and to use composites of composites frequently.

NON-PREMULTIPLICATION PROBLEMS

The analysis above looks pretty bad for the non-premultiplied case, but let's look at it more closely. The bad step is the divide by alpha to return a non-premultiplied color. In the typical case of integer colors and integer alphas, this leads to inaccuracy. And it is not even possible if alpha is 0. But we can sometimes avoid this divide and/or loss of information.

A new alpha of 0 at a composite pixel means (1) that, in the coverage model, neither image I nor J was present at that pixel, or (2) that, in the

[91] I just discovered (5 Nov. 1996) in a correspondence with Marc Levoy that he and Bruce Wallace came up with an equivalent formulation for the nonpremultiplied case in (Wallace 1981), p. 257.
[92] I am loosening the terminology here, temporarily, to extend "sprite-hood" to non-premultiplied images with an alpha.

opacity model, both colors are known but both opacities are 0, or (3) a mixture of these. In case (1), we know that there is no defined color so could store an indication of this in the color channels. In case (2), both colors have to be summarized somehow as one color—for example, an equal mixture of the two is stored. There *must* be a loss of color mixing information here. In case (3), one image is not present and we know the color of the other, so there is no problem.

In the non-0 alpha case, if we have a geometric model of the contributions to a pixel, then we can compute, in the reals, what the mixture of colors at the pixel should be—as opposed to using the integer divide technique above. If all we have is an opacity model at the pixel, then again there must be a loss of color mixing information.

This analysis shows that we can improve the non-premultiplied case but not completely. But, of course, there is no such problem if one is guaranteed to use a sprite for compositing exactly once. More carefully, there is no problem if one is guaranteed to never use the results of a composition for future compositions.

PREMULTIPLICATION PROBLEMS

Is there *anything* wrong with the premultiplied alpha case? Well, yes there is. There are times when one wants the full non-premultiplied color of a pixel. This requires a divide by the corresponding alpha, hence the problems with integer divide and loss of information mentioned above.

So what to do? It seems clear that for reusable sprite objects—particularly reusuable composites of them—the premultiplied case is superior, except for the problem just mentioned. How often does it occur? And how substantial is it when it does occur?

My experience in the graphic arts use of sprites—the Altamira COMPOSER image compositing application, for example—is that the error introduced by the occasional need to divide out alpha is typically so minor as to be unnoticed. In fact, no user has ever noticed it in Altamira COMPOSER to my knowledge. The divide by zero problem never occurs because, by definition, a clear pixel (alpha and all color components equal to 0) does not "exist" so is ignored.

B: IMAGE COMPOSITING FUNDAMENTALS

I am reminded of a division of the geometry-based computer graphics world into what is usually called CAD and, say, CGI. The distinction is that CAD requires *accurate* geometry because it is being used by architects and engineers. CGI is only required to look good. Accuracy can be, and often is, sacrificed in CGI to get a satisfactory look quickly.

The point is that there is a similar division of the sampling-based side of the computer picturing world, based on user type or market. Clearly, accuracy is very important to such users of images as medical doctors and astronomers. But use of images in the graphics arts is much more forgiving. Here, again, the result must be pleasing rather than accurate.

SOME USEFUL APPROXIMATIONS

We derive now some very useful integer approximations for the implied floating point operations in the formulas above. These apply in the case of the very common 8-bit channel—for example, 24-bit color image plus 8-bit alpha.

The integer approximations below are derived from the geometric series

$$a + ar + ar^2 + ar^3 + \ldots = a/(1-r)$$

for $|r| < 1$. We apply the series this way: Let $r = 1/256$. Notice that

$$t/255 \equiv (t/256)/(1-r).$$

Thus, given two numbers a and b, each on $[0, 255]$ and with product t on $[0, 255^2]$, we get $t/255$ on $[0, 255]$—as desired—by using the first two terms of the geometric series:

$$(t \gg 8) + (t \gg 16) + (t \gg 24) + \ldots$$

Notice that

$$(t \gg 8) + (t \gg 16) \equiv ((t \gg 8) + t) \gg 8 \equiv ((t \ll 8) + t) \gg 16.$$

This is captured by the INT_MULT( ) definition below which assumes a and b are each on $[0, 255]$, t is an **int** temporary variable that holds the product $a * b$, which is returned on $[0, 255]$, as if one of a or b were a fraction on [0,1] used to weight the other—for example, as an alpha. In the style of the language C:

#define INT_MULT(a, b, t) $((t) = (a) * (b), ((((t) \gg 8) + (t)) \gg 8))$

We now use the INT_MULT( ) function to define other useful approximations. (We will present an even better macro for it below.)

Classic linear interpolation—or lerp—as it is affectionately called in computer graphics—is defined in floating point below. It is read "lerp p to q by alpha a." a is assumed to lie on $[0, 1]$. Note that $a \equiv 0$ implies p; $a \equiv 1$ implies q.

#define FLOAT_LERP(p, q, a) $((a) * ((q) - (p)) + (p))$

In this integer version t is an **int** temporary variable:

#define INT_LERP(p, q, a, t) $((p) + \text{INT\_MULT}(a, ((q) - (p)), t))$

Premultiplied lerp assumes q has been premultiplied by a.

#define FLOAT_PRELERP(p, q, a) $((p) + (q) - (a) * (p))$

In this integer version t is an **int** temporary variable:

#define INT_PRELERP(p, q, a, t) $((p) + (q) - \text{INT\_MULT}(a, p, t))$

So our formulas for composition (with $'$ (prime) consistently representing premultiplication) become, in the 8-bits per channel case:

8-Bit Compositing Formulas for **over**, Colors Not Premultiplied by Alpha:

$$C' = \text{INT\_LERP}(\text{INT\_MULT}(A, \alpha, t0), B, \beta, t1)$$
$$\gamma = \text{INT\_PRELERP}(\alpha, \beta, \beta, t)$$
$$C = C'/\gamma$$

8-Bit Compositing Formula for **over**, Colors Premultiplied by Alpha:

$$C' = \text{INT\_PRELERP}(A', B', \beta, t)$$

Caution! The approximations above must be used with care. In particular, the case $\alpha = 1$ is a problem. Note that INT_MULT$(255,255, t)$ is 254, not 255. Also note that INT_PRELERP$(255,255,255, t)$ is 256, which is even worse. One-bit errors at other than the high or low end of the range are often tolerable, but not at the extremes. In practice, this is not usually a problem. A typical software loop looks for the special cases of $\alpha = 0$

and $\alpha = 1$, and skips the interpolation computation there. These two cases are so common in imaging that this technique saves much computation. We see from the note above that **it is *important* to check for the $\alpha = 1$ case and avoid the approximation in that case**.

The INT_MULT( ) macro above suffers from 1-bit errors, in about half of the cases. There are better approximations if one does not mind absorbing a little more cost, if special casing is undesirable, or if hardware implementation is the goal. One pointed out to me by colleague John Snyder is to use three terms in the power series approximation: $(t \gg 8) + (t \gg 16) + (t \gg 24)$. This loses no bits, but requires 32-bit arithmetic as written.

Another, pointed out by colleague Jim Blinn, is to use roundoff in the approximation, rather than truncation: $((t \gg 8) + t + 0x80) \gg 8$. This is good, but it still suffers from 1-bit errors in a few cases (24 to be exact). Jim determined that rounding t before shifting got rid of even these errors. So the best macro is this:

#define INT_MULT(a, b, t) $((t) = (a) * (b) + 0x80, ((((t) \gg 8) + (t)) \gg 8))$

at a cost of one additional add. It has no 1-bit errors and can be performed in 16-bit arithmetic.[93] See (Blinn 1994a, Blinn 1994b) for Jim's arguments in support of premultiplied alpha.

IMAGE OBJECTS OR SPRITES

The most important result of using premultiplied alphas is conceptual—the conceptual change from

> **Old Notion:** An image is a *rectilinear* array of pixels. The alpha channel, if any, tells how each pixel is to be treated. Each image pixel has a color that may be masked on or off (or partially on) by the corresponding alpha channel pixel.

to

[93] And can be realized in one register in six Intel instructions! Here they are (also by Jim Blinn and independently by Microsoft's David Jones): mov al,a; mul b; add ax,0x80; add al,ah; adc ah,0; mov r,ah.

SPRITE THEORY

New Notion: An image is a shaped array of pixels with partial transparencies. The alpha channel is intrinsic. The image pixels at transparent pixels (alpha zero) simply do not conceptually exist.

This new notion is captured in the **sprite** object (or image object, as I formerly called it).

I argue strongly for wider adoption of and promulgation of the premultiplied alpha concept that led us to the notion of sprite, and for the elegance of its formulation. It has problems but they are far fewer than those for the alternative.

ACKNOWLEDGEMENT

I keep learning about the subtleties of images and alphas by writing papers such as this. The second-composition problem was never clear to me before. I owe John Bradstreet thanks for a probing and ultimately inspiring question, raised by an early draft of this appendix. Thanks also to Jim Kajiya for his comments on closure, and to Jim Blinn and John Snyder on improved lerp approximations.

BIBLIOGRAPHY

SIGGRAPH'*nn* represents the 19*nn* or 20*nn* SIGGRAPH annual conference, that of the Special Interest Group on computer GRAPHics of the Association for Computing Machinery (ACM).

Beyer, Walter (1964). Traveling Matte Photography and the Blue Screen System. *American Cinematographer*, May 1964, 266. The second of a four-part series. Pre-digital technology.

Blinn, James F. (1994a). Jim Blinn's Corner: Compositing Part 1: Theory. *IEEE Computer Graphics & Applications*, Sept. 1994, 83–87.

——— (1994b). Jim Blinn's Corner: Compositing Part 2: Practice. *IEEE Computer Graphics & Applications*, Nov. 1994, 78–82.

Bracewell, Ron N. (2000). *The Fourier Transform and Its Applications*. 3rd ed. Boston: McGraw-Hill, 2000.

Catmull, Edwin (1978). A Hidden-Surface Algorithm with Anti-Aliasing. *Computer Graphics* 12(1978):6–11. SIGGRAPH'78 Conference Proceedings.

Catmull, Edwin, and Alvy Ray Smith (1980). 3-D Transformations of Images in Scanline Order. *Computer Graphics* 14(1980):279–85. SIGGRAPH'80 Conference Proceedings.

Fielding, Raymond (1972). *The Technique of Special Effects Cinematography*. 3rd ed. Focal/Hastings House, London, 1972, 220-243. Pre-digital technology.

Hume, David. *A Treatise of Human Nature: Being an Attempt to Introduce the Experimental Method of Reasoning into Moral Subjects*. London, 1739.

Oppenheim, Alan V., and Ronald W Schafer (1975). *Digital Signal Processing*. Englewood Cliffs, N.J.: Prentice-Hall, 1975.

Porter, Thomas, and Tom Duff (1984). Compositing Digital Images. *Computer Graphics* 18(1984):253–59. SIGGRAPH'84 Conference Proceedings.

Rabiner, Lawrence R., and Bernard Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, N.J.: Prentice-Hall, 1975.

Smith, Alvy Ray (1978).[94] Color Gamut Transform Pairs. *Computer Graphics* 12(1978):12–19. SIGGRAPH'78 Conference Proceedings.

——— (1981a). Digital Filtering Tutorial for Computer Graphics. Technical (Tech.) Memo 27, Computer Division, Lucasfilm Ltd., Nov. 20, 1981 (revised Mar. 3, 1983). Tutorial notes at SIGGRAPH'83, and SIGGRAPH'84.

[94] Most of my papers and memos are available on <alvyray.com>.

——— (1981b). Digital Filtering Tutorial: Part II. Tech. Memo 44, Computer Division, Lucasfilm Ltd., May 10, 1982 (revised May 11, 1983). Tutorial notes at SIGGRAPH'83, and SIGGRAPH'84.

——— (1982a). Analysis of the Color-Difference Technique. Tech. Memo 30, Computer Division, Lucasfilm Ltd., Mar. 29, 1982.

——— (1982b). Math of Mattings. Tech. Memo 32, Computer Division, Lucasfilm Ltd., Apr. 2, 1982. Reissue of an unnumbered tech. memo of Dec. 30, 1980.

——— (1982c). Digital Filtering Tutorial, Part II. Tech. Memo 44, Computer Division, Lucasfilm Ltd., May 10, 1982 (revised May 11, 1983).

——— (1983). Spline Tutorial Notes. Tech. Memo 77, Computer Division, Lucasfilm Ltd., May 8, 1983. Tutorial notes at SIGGRAPH'83, and SIGGRAPH'84.

——— (1988a). VAIL—The ICEMAN Volume and Imaging Language. Tech. Memo 203, Pixar, Marin County, Calif., June 29, 1988, Nov. 28, 1988, and Jan. 4, 1989, respectively, for chapters 1, 2, and 3.

——— (1988b). Geometry and Imaging. *Computer Graphics World*, Nov., 90–94.

——— (1989). Two Useful Box Routines. Tech. Memo 216, Pixar, Marin County, Calif., Dec. 26, 1989.

——— (1990). An RGBA Window System, Featuring Prioritized Full Color Images of Arbitrary Shape and Transparency and a Novel Picking Scheme for Them. Tech. Memo 221, Pixar, Marin County, Calif., July 13, 1990.

——— (1995a). Image Compositing Fundamentals. Tech. Memo 4, Microsoft, Redmond, Wash., June 1995 (revised Aug. 15, 1995). See appendix B.

——— (1995b). A Sprite Theory of Image Computing, Tech. Memo 5, Microsoft, Redmond, Wash., July 17, 1995.

——— (1995c). A Pixel Is *Not* a Little Square, a Pixel Is *Not* a Little Square, a Pixel Is *Not* a Little Square! (and a Voxel Is Not a Little Cube). Tech. Memo 6, Microsoft, Redmond, Wash., July 17, 1995. See appendix A.

——— (1995d). Alpha and the History of Digital Compositing. Tech. Memo 7, Microsoft, Redmond, Wash., Aug. 15, 1995.

——— (1996). Image Sprite Functions. Tech. Memo 12, Microsoft, Redmond, Wash., June 24, 1996.

Smith, Alvy Ray., and Eric R. Lyons (1996). Hwb—A More Intuitive Hue-Based Color Model. *Journal of Graphics Tools*, 1, 3–17.

Wallace, Bruce A. (1981). Merging and Transformation of Raster Images for Cartoon Animation. *Computer Graphics* 15(1981):253–62. SIGGRAPH'81 Conference Proceedings.

ILLUSTRATIONS

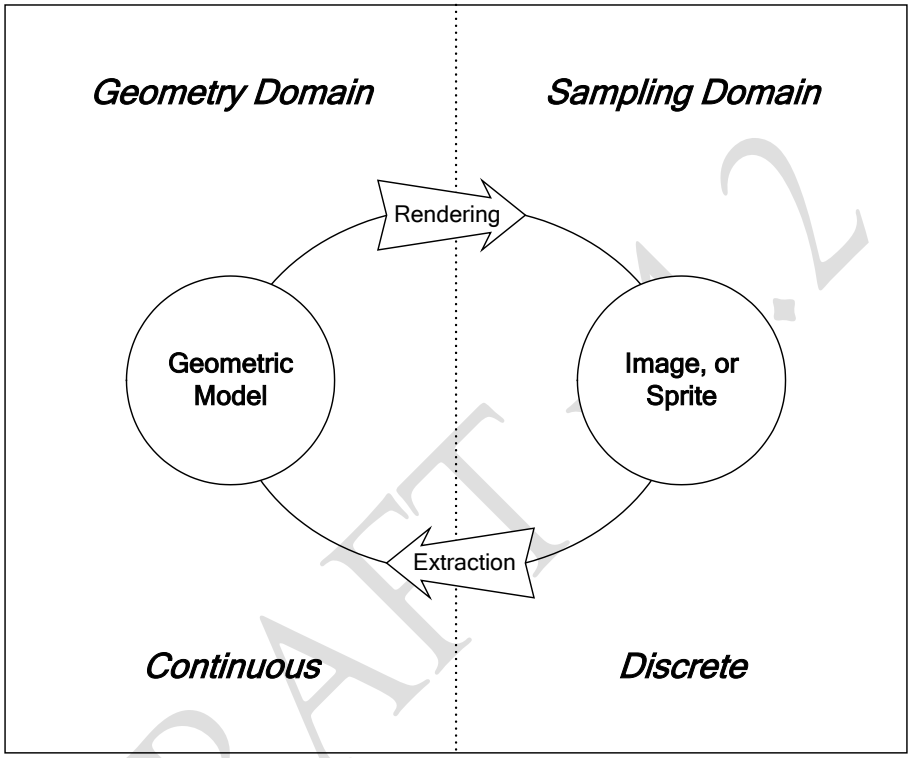


Figure 1. Geometry vs sampling

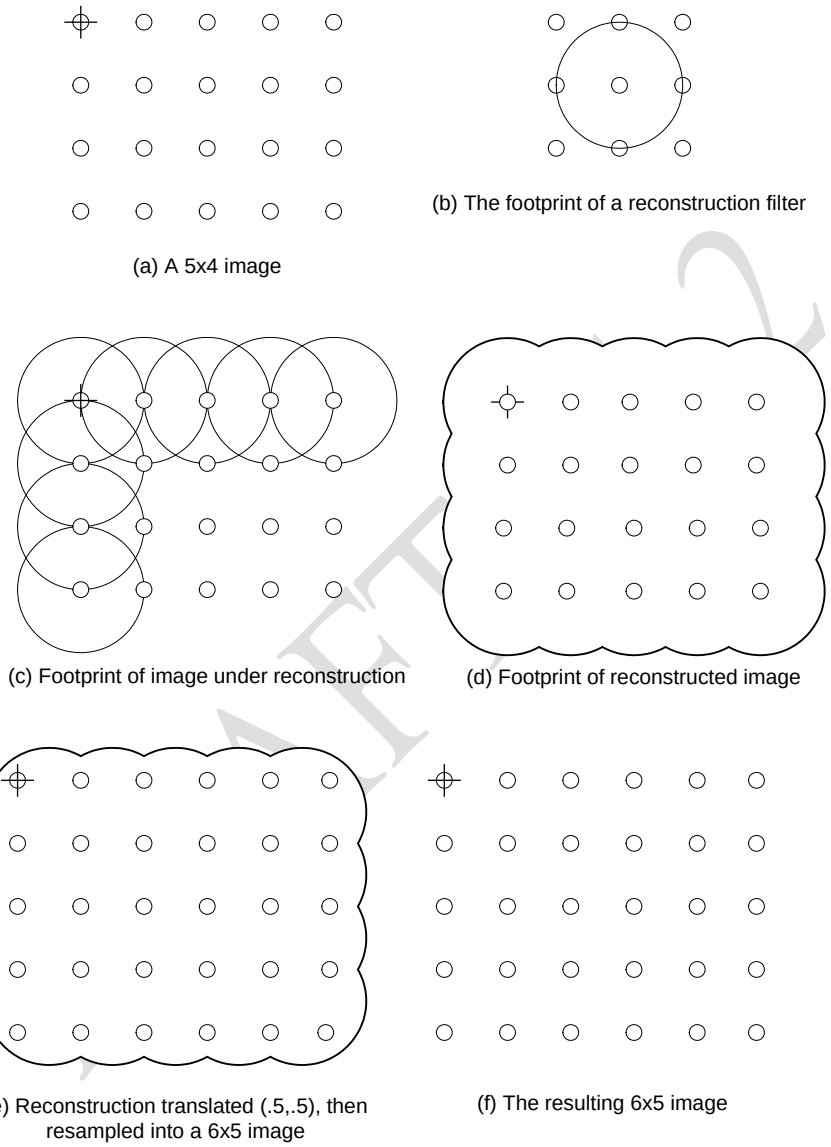
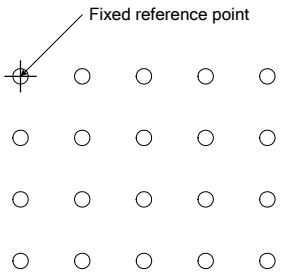
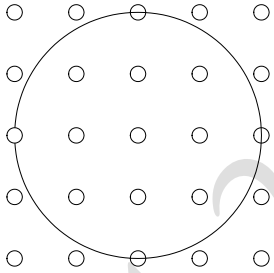


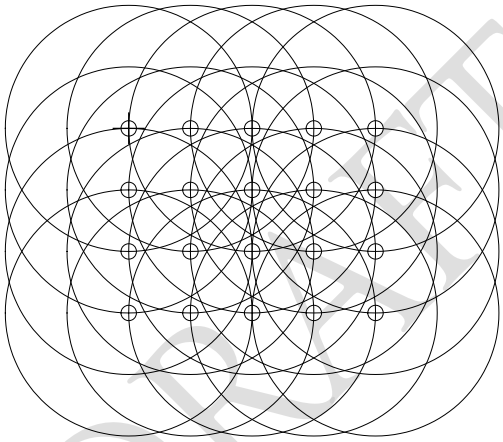
Figure 2. To do geometry on images: Reconstruct, transform, resample



(a) A 5x4 image.

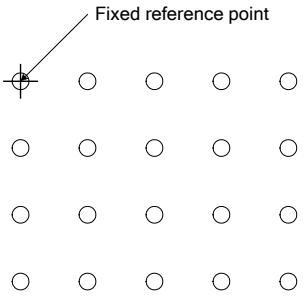


(b) The footprint of a reconstruction filter.
A cubic, or windowed sinc, for example.



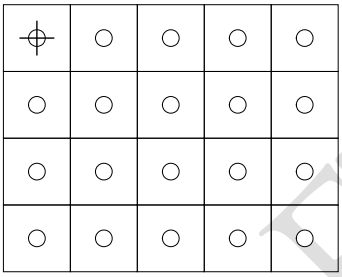
(c) Footprint of reconstructed image.
Typical high quality reconstruction.
Would be resampled into 7x6 image.

Figure 3. Excellent reconstruction



(a) A 5x4 image.

(b) The footprint of a reconstruction filter.
A simple box filter, for example.



(c) Footprint of reconstructed image.
The worst case: low quality reconstruction.

Figure 4. Crude reconstruction