A Sprite Theory of Image Computing

Technical Memo 5

Alvy Ray Smith
July 17, 1995

TABLE OF CONTENTS

CHAPTER 1: OVERVIEW

Introduction

This paper is an introduction to a theory of image computing. The theory, or model, underlies everything I say or think about images and imaging. It is also the implicit model behind the software application, Altamira Composer, that Microsoft inherited in its purchase of my company Altamira Software Corporation last year.

I have found myself involved in many different conversations within Microsoft, including many related to the adaptation of substantial portions of Altamira Composer source code for future Microsoft products. It is difficult to carry on these discussions without a shared understanding of the concepts presented here. This document should also serve to ease the task of those attempting to learn the Altamira Composer code. And aspects of the model will greatly influence the highly generalized media model of the Media Foundation that Nicholas Clay and I (and many others) are currently designing.

Much of what appears here is a reworking of ideas first presented in [Smith89a], modified by experience using the ideas in an actual application, Altamira Composer.

Origins

The primary idea behind the model to be presented here, the Altamira sprite theory or model of image computing, was that one was needed at all. It came from my realization, in the 1980s, that the reason the “image processing” market had not taken off was that it had no center—it was undefined. As opposed to the 2D geometry market—also known as desktop publishing—which was founded on the careful definition of 2D geometry embodied in the PostScript language, there was no 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. So I spent about a year at Pixar defining a language, called IceMan, to accomplish for images what PostScript did for 2D geometry. Many of these concepts, but not the IceMan language itself, I embodied in a prototype application that eventually became Altamira Composer. The concepts grew and matured in Altamira Composer, and they are those presented here[1].

But I could not have come up with the concepts in the first place if there had not been two important ideas already in place: the alpha channel and matting

[1] The IceMan language remained with Pixar when I left to form Altamira Software and is proprietarily Pixar’s. However, the concepts embodied in Altamira Composer may be used freely by Microsoft. Altamira Composer is not based on a language, but it does share some definitions with IceMan, such as image, box, and point.

"algebra"[2]—particularly, premultiplied alpha. I and my colleague Ed Catmull came up with the concept of the alpha channel in the 1970s at the New York Institute of Technology. It has become so standard in computer graphics that I do not think I have to explain it. It is one of those concepts that one can hardly believe had to be invented. The crucial distinction involved in the invention was not that two images could be combined by use of linear interpolation under control of a mixing factor, typically denoted α. This notion, image composition, was obvious and used often even in the early days of computer graphics. The important leap was to append alpha to *every* pixel as a component, the alpha channel, of a pixel just as fundamental to it as its color components. Thus one began to think of opacity as an intrinsic part of an image rather than as a separate notion.

It is not hard to understand why we didn't leap to the concept of the integral alpha before we did. Recall that at the time memory was still very expensive. A video framebuffer, 640x480x8 bits, cost $60,000-$80,000 (cf current price of about $100). So an RGB framebuffer cost about $200,000 and an RGBA framebuffer (with an alpha channel storage) cost about $250,000 (in 1975 dollars)[3]. It was nontrivial to increase memory usage by 25%. And we were the only facility in the world that had even an 8-bit framebuffer, and we were first for a long time with a 24-bit one and a 32-bit one. Nevertheless, once we had the hardware, we quickly got to the software notion.

Although we added the alpha channel to our thoughts and computations and hardware, we still did not fully understand it. It wasn't until my Lucasfilm/Pixar colleagues Tom Porter and Tom Duff invented the matting algebra (see [PorterDuff84]) that the confusion between premultiplied and not-premultiplied alpha was made explicit and cleared up. I have discussed the fundamental compositing operator **over** and the premultiplied alpha concept in [Smith95], where the case for premultiplied alpha is improved and shown to be strong.

The most important idea that became possible with the understanding of premultiplied alpha is the *shaped image* —that is, a non-rectangular image. I point out in [Smith95] that because premultiplication by alpha completely clears transparent pixels of any current or future information, they cease to exist conceptually (although they still might continue to occupy memory). A pixel with alpha equal 0 requires, in the premultiplied alpha case, that all of its color channels be 0 too. And once one ceases to think of empty pixels, it is an easy leap to shaped images.

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 market-

[2] Although often called a matting *algebra*, it is not really an algebra by any usual mathematical definition of that term.

[3] That's about $1 million in 1995 dollars!

ing. This got confused with "layers" in the market, a restricted and rigid concept[4] that can be simulated with image objects but not vice versa. So here at Microsoft we are 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 kept the confusion with layers at bay[5].

So it was alpha intimately in the pixel—the alpha channel—and premultiplied alpha—with transparent pixels forever superfluous and hence conceptually nonexistent—that led eventually to the shaped image, or image object, 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 or image object—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 below, 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"—that connotes doing things to a static monolithic rectangular image—to "image composition"—that connotes a space full of floating shaped, 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 interaction with images. Most of the operations have to be rethought, however, to handle alpha and shape information.

Sampling v Geometry

One contribution of the sprite theory of image computation is a careful delineation between imaging, or sampling, and geometry. I have drawn this fundamental distinction many times (eg, [Smith88]), but it bears repeating because the confusion is still common. See Figure 1.

There are two different ways to make pictures with computers. The one most popularly understood is that based on geometry, often called "computer graphics" or "CGI" or "3D synthesis". This includes *Jurassic Park* graphics and the soon-to-be-released *Toy Story* from Pixar and Disney. The other is based on sam-

---

[4] To get a feel for the limitations of the layer concept, try to generalize it to 3D. Freely floating objects easily generalize, but layers don't. Furthermore, layers usually connote a stack of rectangles, 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. There is no conceptual problem with having two sprites at the same depth but in layer terminology, one would have to assign the two sprites to the pre-existing layer somehow to model the concept. This is unnecessary machinery.

[5] 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 PC 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". The rest is, as they say, history.

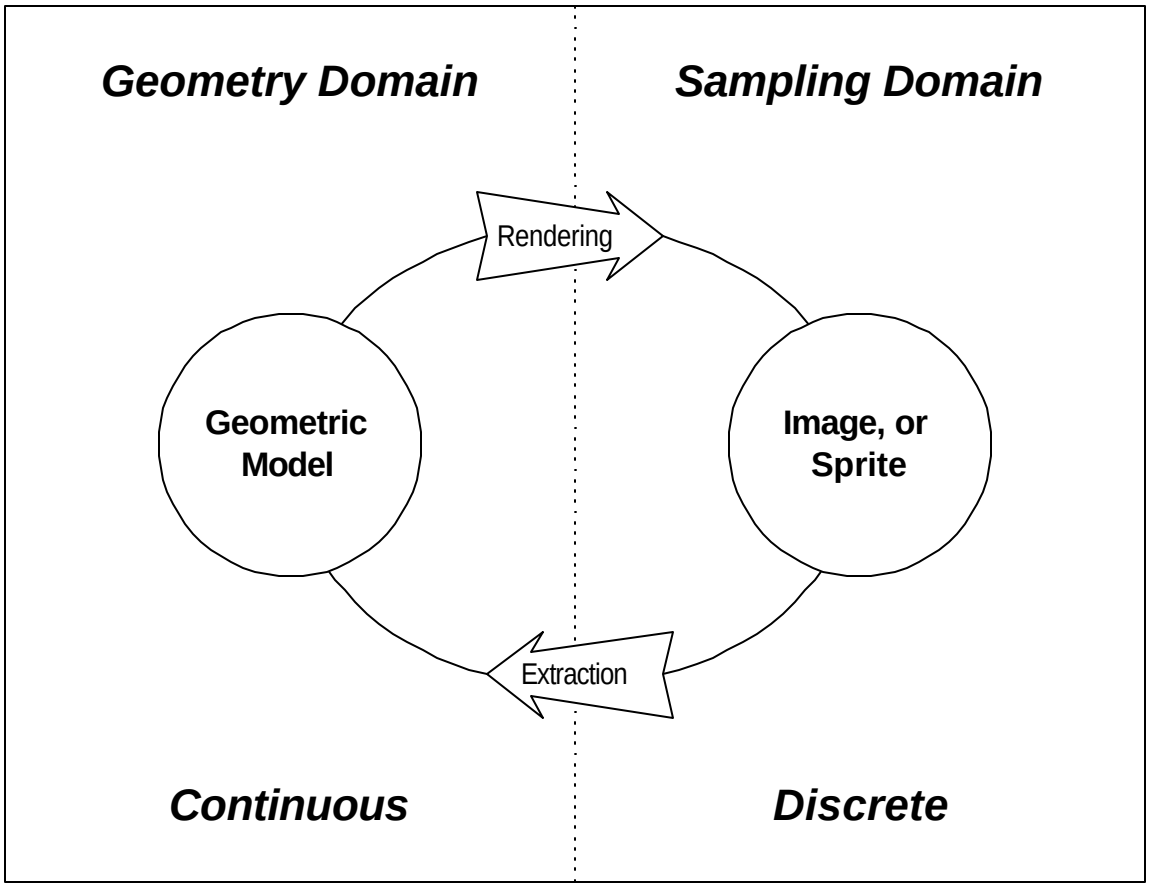


Figure 1

pling theory, often called “image processing” or “imaging”. The digital pictures coming back from the Voyager planetary flybys are of this type, as are digitized photographs, and digital video. The Visible Human from the National Library of Medicine is another excellent example. In the sampling-based way, there is no geometry involved at all. A classic pair of apps that implicitly make the distinction are MacPaint (imaging) and MacDraw (geometry). This also shows that computer graphics and image processing are sloppy terms, since paint programs are nearly always included in the former but not in the latter.

The sampling v geometry distinction is fundamental. The two paths, *geometry-based* and *sampling-based* (we will also sometimes say *image-based*), 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 if true[6] then it only applies to the geometry half of computer graph-

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

ics (now extending the term to include all ways of making pictures on computers).

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). This conversion step is called *rendering*. Since both approaches to picture making with computers result in an array of pixels, many nontechnical people cannot distinguish the two processes.

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 often 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 nearby are these [a correction or criticism is in square brackets]:

- A pixel is modeled as a little geometric square[7] [a pixel is a point sample]
- A pixel is located on the half pixels [samples are array elements with indices as "location"; there is no such thing as a half pixel—comes from the little square model]
- A monitor has non-square pixels [again, pixels are point samples; the correct notion is *pixel spacing ratio* (PSR); the notion being conveyed is that the sampling distance in the horizontal dimension is different from that in the vertical—ie, the PSR is not 1, or "non-square pixel spacing"]
- Images have regions of interest [usually means a geometrically defined region; the mapping to a sampled image is undefined, it being assumed obvious; it never is; the alpha channel nearly always captures the notion accurately and with more generality]
- An image is a rectangle or rectangular picture [an image is a rectangular array of samples; it does not reconstruct, by the Sampling Theorem, into a rectangle or rectangular picture[8], in any but the dumbest use of that theorem (with the worst[9] reconstruction filter, a box)]

[7] 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 simplying model with the pixel itself.

[8] See the section below entitled Continuous Operators on Discrete Sprites and its Figure 2.

[9] Well, the worst is really no filter at all—point sampling.

Creative Space v Display Space

Another fundamental distinction that the following 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. One models in 3D unbounded space with abstract geometrical objects. One displays views of this *creative space* that are 2D and restricted to the size allowed by some output medium such as 35mm film or NTSC video. The choice of *display space* is separate from the creation or modeling in creative space. It is, in fact, a separate creative step. 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 confuse the two. For example, most popular imaging applications equate display space to the rectangular image being edited. One opens an image (meaning rectangular 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 modeled as 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, aligned against one another, and so forth. In our 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, ie, with the renderer suitable to the type of the object.

Definition of Image

> 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. Our definition is still quite broad. In an actual implementation of our model, only certain available data types—eg, integers, floats, bytes—are permitted for representing channels, hence images.

An image is *n*D, 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. 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* for 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 rectangular parallelepiped. And so forth. Clearly, non-rectilinear images are necessary—eg, as brushes in a paint program. The model handles arbitrary shape with an imaging notion, not a geometrical one. Non-rectilinear shape is 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*. Where a matte is 0, the corresponding image can be considered to not exist (or be transparent). Where it is 1[10], the image is opaque. Where fractional between 0 and 1, the corresponding image is partially transparent. This powerful notion allows arbitrary shapes including disconnected components and components with holes.

An image has pixels of identical type, and its pixels have channelvalues of identical type. 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. A pixel with more than five or less than five channels is not allowed in an image of these pixels. An image may also be thought of as a list of **channel**s, where the i^{th} channel of the image is an array of all the i^{th} channelvalues of the pixels comprising the image. The model provides utilities for combining channels into a thicker (in the sense of more channels) image, or for extracting channels 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.

To represent channels with different types, they are represented as separate images and these are bound together with a higher order concept (such as the “imagestruct” in Altamira Composer).

[10] We use a floating point number between 0. and 1. for alpha for convenience only. It is often realized in an 8-bit unsigned byte with values between 0 and 255.

An image consists of samples. A fundamental assumption of the model is that each channel of an image can be reconstructed, using the Sampling Theorem, to retrieve the continuum which it represents—eg, 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 4x4 matrix is 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 an application, unless it 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 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 640x480 samples, must have the green and blue channels also at sizes 640x480. 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. The model requires that images at different resolutions be treated as separate images[11]. An application binds them into a single entity as a higher-order structure. This includes so-called multi-resolution representations of images. Sometimes it is possible to bitpack, or otherwise encode, a higher resolution image into a lower resolution data type which then *is* permitted to be a channel at the lower resolution. This works if the higher resolution is an integer multiple in each dimension of the lower.

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

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 (as is common), then they may of course be treated as channels of a single image.

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—ie, on the nodes of a rectilinear grid. So hexagonal sampling grids, for example, are not used by the model. Again, an application can elect to disregard this assumption. As detailed below, the model takes a rectilinear subset of the integer grid to be the support of an image. The assignment of different sampling rates to different dimensions is application dependent.

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

Definition of Sprite and Shape

The full generality of an image is not actually implemented in Altamira Composer. Instead the special case of a 2D RGBA *sprite* is emphasized. That is, all image objects in Altamira Composer are assumed to have exactly two dimensions and four channels, corresponding normally to three color channels (Red,

Green, and Blue) and one matte or alpha channel (Alpha). Furthermore, the alpha channel is assumed to be premultiplied. That is, the color channels are assumed normally to have been premultiplied by the alpha channel. Practically this means that no number in a color channel can exceed the number in the alpha channel in the corresponding position. In particular, transparent pixels (alpha 0) have all three color channelvalues 0 also. Such a pixel is referred to as *clear*—that is, a completely transparent black pixel is a clear pixel. As argued above 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 summary:

> A **sprite** is an RGBA image where the RGB channels are assumed to represent color premultiplied by the A, or alpha, channel.

Altamira Composer uses more general images and non-premultiplied alphas when necessary or convenient, but the most important object is the sprite.

Sprites have shape, which we carefully define as follows:

> The **shape** of a sprite is exactly the subset of its pixels with non-0 alpha.

Thus the shape of a sprite (or image in general) is defined in sampling terms, not geometrical. It is sometimes convenient—eg, succinct—to describe a shape geometrically, but what is *always* meant is this: The geometric "shape" is rendered into an image in an alpha channel, at which point it becomes a shape by our definition. Notice that shape includes opacity information as well.

Coordinate Systems

The notion of coordinate system usually comes equipped with real space connotations when applied to images. An image does not have a real coordinate system. Instead, its pixel "coordinates" 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][12]. 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 extremely easy. 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

[12] Using C-like array notation.

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 object itself.

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 app, but it is a remarkably simple one[13]. There is a requirement, however, that images and sprites be thought of as arrays and indexed in the standard way.

Continuous Operators on Discrete Sprites

There are two ways to slide a sprite around in this creative space. As usual, the two conceptions come from the two worlds, continuous and discrete, and we shall be careful to 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 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, that we have not yet detailed).

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 about images with no further explanation, as if it were obvious what it means. 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 our discrete images and sprites.

Our model derives directly from sampling theory. One of the most amazing theorems we have is the Sampling Theorem. The truth it conveys is what makes our business of computer graphics possible. Similarly it is what makes animation possible, print possible, and digital audio possible. This theorem states, in crude

[13] 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.

terms, that a continuous thing, with an infinity of points, can be represented[14] by a discrete thing, with a finite set of points called samples, plus a reconstruction filter, which is a continuous thing again. One way to think of this is that the infinite information of the given continuous entity is somehow summarized in the much simpler, but still infinite, reconstruction filter. What makes all this work is that many of our display devices (including audio "displays") automatically reconstruct point samples by their very nature. That is, the reconstruction filter does not have to be explicitly applied because the display device automatically does it. For instance, in a cathode ray tube the photon emission of screen phosphors serves to spread a point sample, supplied as a change of voltage to the tube, as if a reconstruction filter had been applied to it as required by the Sampling Theorem to retrieve the continuous from its discrete representation. That is, discrete image memory samples driving a CRT appear to cause display of a continuous image to us[15]. So here is how to think of a continuous operator on a discrete sprite (cf Figure 2): 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. Now that we know exactly what 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. It is important to notice what we have *not* said. We have not ever used the word rectangle. In fact, the edge of a reconstructed sprite is hardly ever accurately represented by a rectangle. See Figure 2 for details. We 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. We have not specified what the reconstruction filter is[16]. The correct one, according to the Sampling Theorem, is the so-called "sinc" filter[17], but this is infinite in spatial extent and

---

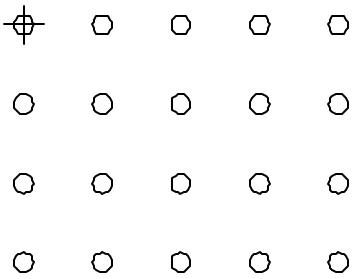
[14] Under certain restrictions, that we shall not be overly concerned about here, that ensure the continuous entity does not dramatically "change" more often than the discrete samples are taken.

[15] By the way, the phosphors of a display should not be confused with a pixel. There is only an approximate spatial relationship between sets of phosphors and a pixel that drives it. There is not a one-to-one mapping.

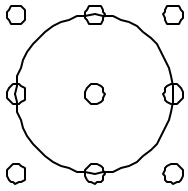
[16] The class of reconstruction filters represented by the footprint of Figure 2b is a simple one. The reconstruction filters of Altamira Composer, for example, are generally bicubic which means that their footprint extends across two sampling intervals in each direction, not just one as shown in the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

[17] The sinc filter is generated from the function $\sin x / x$.

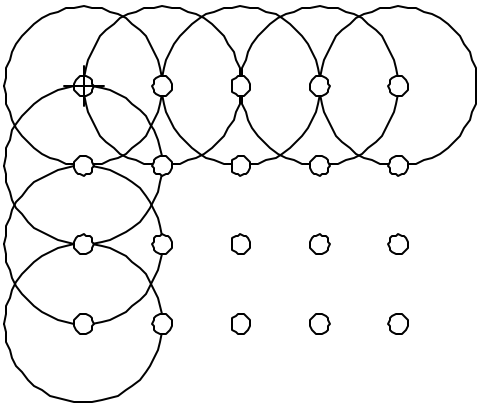
thus not practical to use. In practice, a variety of approximations to the sinc filter, with finite spatial extent, are used. The worst approximation used is called a box filter, but this is the only one ever used that gives a reconstruction with a rectangular boundary. Altamira Composer generally uses much more sophisticated cubic filters. The filter used is an implementation detail, but we highly discourage the use of box filters to ensure high quality results.



(a) A 5x4 image

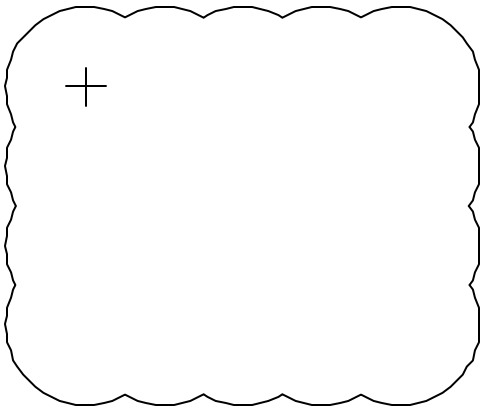


(b) The footprint of a reconstruction filter

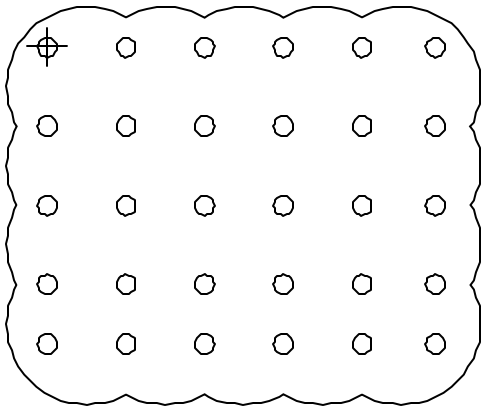


(c) Footprint of image under reconstruction

FIGURE 2



(d) Footprint of reconstructed image



(e) Reconstruction translated (.5,.5), then resampled into a 6x5 image

CHAPTER 2: BOX ALGEBRA

Box Algebra

Support calculations comprise a major component of image computation, just as address or pointer calculations are a common component of ordinary computation. The Altamira model provides a rich arithmetic, called the *box algebra*[18], for support calculations. Perhaps the most elaborate example of how far one can get in imaging with just box algebra is described in [Smith90] that defines the theory behind the compositing and display engine ultimately used in Altamira Composer.

By convention, the pixels of an image or sprite always reside on the integer grid. Each point of the *n*D integer grid is described by an *n*-tuple of integer coordinates, an *n*D integer *point*. The integer point associated with a pixel in an image can be thought of as its address in integer coordinate space—not to be confused with a physical memory location allocated to hold the pixel. The *support* of an image is the set of addresses of its pixels. The rectangular support of an image can be represented by its minimally enclosing *box*. We again carefully distinguish a discrete concept from a geometrical one. 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 handy sometimes so points and boxes with real (**float**) coordinates are allowed.

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.

Support

When an image or sprite is declared, its pixels are located at all *n*D integer points in the rectangular set of points extending from $\mathbf{0} \equiv (0, 0, \ldots, 0)$ to $(size_0 - 1\ size_1 - 1, \ldots, size_{n-1} - 1)$, where $size_i$ is the number of pixels of the image in dimension *i*. Any rectangular 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 rectangular set of integer points which index the pixels of an image or subimage form its support. Similarly for sprite and *subsprite*.

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. The model 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

[18] I attempt to justify the use of the word “algebra” in a later section.

the 2D case, because this is what Altamira Composer actually implements. Generalization to higher dimension is not difficult.

A **point** has two elements, called *coordinates*, of type **int**[19]. A **floatpoint** has two coordinates of type **float**. The coordinates are assumed to be named *x* and *y*. For convenience, given a (**float**)**point** *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 a *meta-notation*.

There is a special **point** called **zeropoint** always available that is simply a point with two 0 coordinates. Altamira Composer implements this as a method **point** ZeroPoint(**void**). A useful method is **boolean** IsZeroPoint(**point** p). A **boolean**, of course, has two values **true** or **false** and is realized in the Windows development lexicon with type BOOL that has 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* in both coordinates. We do *not* represent this in meta-notation with *p* == *q*. See the relational operators below for this case.

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

A **point** can be cast to a **floatpoint**, and vice versa. The action is the one expected by one 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 truncate 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 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—ie, the integer just right of 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 **point** UnderPoint(**floatpoint** p) and **point** OverPoint(**floatpoint** p).

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

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

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)	- (unary) 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 == q	**booleanpoint** EQPoint(**point** p, **point** q)	== coordinate-wise
p != q	**booleanpoint** NEPoint(**point** p, **point** q)	!= coordinate-wise
p < q	**booleanpoint** LTPoint(**point** p, **point** q)	< coordinate-wise
p <= q	**booleanpoint** LEPoint(**point** p, **point** q)	<= coordinate-wise
p > q	**booleanpoint** GTPoint(**point** p, **point** q)	> coordinate-wise
p >= q	**booleanpoint** GEPoint(**point** p, **point** q)	>= coordinate-wise
m ? p : q	**booleanpoint** WherePoint(**booleanpoint** m, **point** p, **point** q)	?: coordinate-wise

Boxes

The most common support structure required in imaging is the box. Sometimes an entire image computation can be performed on its box alone, without ever referring to the actual pixel values of the image except for trivial data copying. See Chapter 3, An Informative Example.

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**. We 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*—*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 is different 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.

A **box** can be *invalid*—for example, if its *xmin* is greater than its *xmax*. 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 so did not implement it[20]. 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. It is not clear whether a **floatbox** *b* with *b.min* = *b.max*—ie, a "rectangle" consisting of a single point—should be considered a valid rectangle. See the discussion of box algebra in a later section.

The equality of two boxes is determined with **boolean** EqualBox(**box** b, **box** c)[21], that is **true** if *b* and *c* represent exactly the same set of pixels—ie, 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)

[20] In fact, very little functionality for **floatpoint**s and **floatbox**s is implemented in Altamira Composer. We implemented just those methods we actually needed.

[21] Altamira Composer actually implements many **boolean**s as **int**s, but this is a minor implementation detail.

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 $p \parallel q$, and $\parallel$ is called the *box operator*. The fourth one assumes *p* and *q* are the minpoint and maxpoint, respectively, of the box constructed. It can create an invalid box since no checking is done.

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* = b01 yields the upper right corner. 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 **point** BoxCenter(**box** b), that truncates, or **floatpoint** FloatBoxCenter(**box** b), that doesn't. 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** is not implemented.

A **box** may be cast to a **floatbox** and vice versa. In Altamira Composer, the methods are **box** IntBox(**floatbox** b), that truncates, **box** RoundIntBox(**floatbox** b), that rounds, and **floatbox** FloatBox(**box** b).

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—or equivalently $\lfloor b \rfloor$ and $\lceil b \rceil$. In meta-notation, the definitions are $\lfloor b \rfloor = \lceil \underline{b} \rceil \parallel \lfloor \bar{b} \rfloor$ and $\lceil b \rceil = \lfloor \underline{b} \rfloor \parallel \lceil \bar{b} \rceil$, respectively.

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*—ie, the two boxes don't intersect. The intersection *a* of box *b* and box *c* is computed in *x* 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* && *c*, and && is called the *intersection operator*. The **floatbox** version is not implemented in Altamira Composer.

UnionBox() returns the minimal box enclosing the two box operands. Meta-notation for union $b \parallel c$ uses the box operator introduced earlier for construction of a box from two points. This use is consistent with the former use if a **point** is thought of as a degenerate **box**. The **floatbox** version is not implemented in Altamira Composer.

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

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

or by $[\underline{b}+\underline{c}] \parallel [\bar{b}+\bar{c}]$ 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* so 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)

returns **true** if box *b* lies totally within box *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)

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 $b + p = (\underline{b} + p) \,\|\, (\bar{b} + p)$. 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

$$b \textbf{ op } p = (\underline{b} \textbf{ op } p) \,\|\, (\bar{b} \textbf{ op } p)$$

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

$$b \textbf{ op } c = (\underline{b} \textbf{ op } \underline{c}) \,\|\, (\bar{b} \textbf{ op } \bar{c}).$$

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

MulFloatBox(), *b* ∗ *p*, is defined similarly. In this case, it is the **box** version that is 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 is 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 (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 return value is a 4-bit flag with one bit corresponding to each returned box. Each bit is 1 if the corresponding box is valid.

Special Box Routines

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 [Smith89b]. I invented two nonobvious functions that greatly eased this problem:

```
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. Let *p* be the offset. Then, in meta-notation, the computation is (*t* is a temporary **box** variable):

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

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 && b;
if( !ValidBox(t)) return false;
*S = t;
*D = t + D.min - S.min;
return true;
```

Alternative Algebra[22]

I have called this a box algebra. How close is it really to an algebra? In this section, I will show that || and && have the right properties for the additive and multiplicative operators of an algebra. Then a complement operator will be defined, and identity elements for the additive and multiplicative operators. Then we will revisit the problem. The purpose of this argument is to show that a complete mathematical algebra can be defined on boxes, but the benefit of doing so is questionable, considering the additional machinery that has to be put in place.

First, let's introduce a *complement operator* ~ by the definition, for **box** *b*, of ~*b* to be the box obtained by swapping *b.min* with *b.max*. This would force a valid box to be invalid in the interpretation given so far, but in this section we change the interpretation of the complement of a valid box to be, not an invalid box, but rather a representation of the set complement of the valid box in toroidal 2D space. Furthermore, it is understood that the values stored in a minmax pair with

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

a maximum value less than the minimum value are *not* included in the interval represented by the minmax pair. That is, a valid minmax pair includes its endpoints; an invalid one does not. This is required to make the intersection of a box and its complement the special **emptybox**, a box that represents enclosure of no space.

So 2D space is assumed to be toroidally connected here. It will be assumed infinite for this discussion—that is, the toroid passes through infinity—but it could be finite and still work. An invalid box is no longer thought of as one oriented from minimum to maximum in the wrong direction, but rather as one which passes from minimum, through infinity (or at least around the toroid), to maximum in the same direction as a valid box.

Notice, however, that there is a problem with one-point boxes (minpoint equal to maxpoint). There is no way to tell if a minmax pair represents the box or its complement. Assume for the remainder of this discussion that some mechanism is established for handling this special case.

We need another special box, the **universalbox** that represents enclosure of all of 2D space. Both **universalbox** and **emptybox** are considered valid. A *normal* box is any valid or invalid box other than **universalbox** or **emptybox**. **universalbox** and **emptybox** will be the identities for the && and || operators, respectively.

Notice that the || operator is commutative. That is, $b \parallel c \equiv c \parallel b$, for boxes b and c, because finding minima and maxima is not dependent on order. It is also idempotent, $b \parallel b \equiv b$.

The || operator can also be shown to be associative. The only difficulty is showing that invalid box operands do not destroy the property. The definition of || applied to $w \equiv (a \parallel b) \parallel c$, 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, || is too, and the proof does not depend on whether the minmax pairs are valid.

The && operator is commutative by the same argument as for ||. Thus $b \,\&\&\, c \equiv c \,\&\&\, b$. It is also idempotent: $b \,\&\&\, b \equiv b$.

The && operator can also be shown to be associative. The only difficulty again is showing that invalid box operands do not destroy the property. The definition of && applied to $w \equiv (a \,\&\&\, b) \,\&\&\, c$, 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, and the proof does not depend on whether the minmax pairs are valid.

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

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

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 &&.

Now consider the complement operator ~. It has the property $\sim(\sim b) \equiv b$. It can be shown that the following duality laws hold for ~, ||, and &&: $\sim(b$ && $c) \equiv \sim b \parallel \sim c$ and $\sim(b \parallel c) \equiv \sim b$ && $\sim c$. Minmax arguments similar to those for associativity and distributivity can be used to prove these.

So ~, &&, and || satisfy nearly all the requirements for a Boolean algebra on boxes, except for the existence of identity elements for && and ||. But b && **universalbox** $\equiv b$ and $b \parallel$ **emptybox** $\equiv b$, so **universalbox** and **emptybox** are these identity elements.

Now we do the hard part. We redefine &&, and then || similarly, to handle the new interpretation of boxes. Thus the && intersection operator definition is changed so that the two sets represented by the operands are intersected as sets, it being understood that the set represented by an invalid box passes through infinity and is, in fact, the set complement of the set enclosed by the invalid box. We have to handle the case of a box representing partial infinity—that passes through infinity in only one dimension. If the resulting set intersection is empty or in any other way cannot be represented by a normal box, then the result is defined to be the **emptybox**.

The algorithm for the redefined && follows, for operand boxes b and c and return box w. It is applied to each dimension i (x or y).

```
if( ValidBox( b) && ValidBox( c)) {          // Both boxes valid
    if( c.i > b.i || c.i < b.i ) return emptybox.i;
    w.i  = ( c.i > b.i ) ? c.i : b.i;
    w.i  = ( c.i < b.i ) ? c.i : b.i;
}
else if( !ValidBox( b) && ValidBox( c)) {    // One box valid, the other not
    if( ( c.i > b.i && c.i < b.i ) || ( c.i ≤ b.i && c.i ≥ b.i )) return emptybox.i;
    w.i = ( c.i < b.i || c.i > b.i ) ? c.i : b.i;
    w.i = ( c.i > b.i || c.i < b.i ) ? c.i : b.i;
}
else {                                       // Both boxes invalid
```

```
    if( c.i ≤ b̄.i || c̄.i ≥ b.i ) return emptybox.i;
    w.i = ( c.i < b.i ) ? b.i : c.i ;
    w̄.i = ( c̄.i > b̄.i ) ? b̄.i : c̄.i ;
}
return w.i;
```

The || operator can be redefined with a similarly complex algorithm. I consider these algorithms too expensive to justify implementing a full Boolean algebra. Furthermore, the baggage required for representing **universalbox** and **emptybox**, partially **universalbox** and partially **emptybox**, and the complement of one-point boxes is not considered justified.

I will continue to refer the “box algebra”, knowing that we could, if necessary, extend it to an algebra but not doing so for the reasons just given.

CHAPTER 3: 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 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[23] 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—eg, at the "imagestruct" level in Altamira Composer. 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 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.

[23] We do not attempt to extend this "algebra" to a mathematically complete one, as we did for the box algebra.

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—eg, 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 exist for testing equality of channels and channeltypes.

Color

It is not necessary to discuss color before defining pixels and images with more care, but the RGBA definition of 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**[24]. **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. See [SmithLyons92] for details on the newer, simpler Hwb color representation. 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—eg, **rgbcolor*** HwbToRGB(**rgbcolor*** phwb)[25].

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

[24] Altamira Composer actually uses names RGBColorType, RGBAColorType, RGBFloatColortype, and RGBAFloatColorType, respectively.

[25] These are not actually implemented in Altamira Composer, but the app does contain the conversions indicated, in a different guise—eg, between pixels.

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).

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[26].

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 we 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 for example: 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).

Pixels can be converted to colors and vice versa—*rgb* = *px*, or *px* = *rgb*, for example, in meta-notation, 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 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**.

[26] For clarity, I omit another argument CompStruct* *pComp* that Altamira Composer threads through most routines. I do this as a general rule in this paper.

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 return 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 Chapter 1, I defined the image as a rectangular 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.

There are two popular ways to implement an image consistent with the Altamira 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 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. The principal advantage of this method is the ease with which the ply of an image can be changed—from single channel rectangular 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[27] **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 (see [Smith95] for details).

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** (0x0). 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. See [Smith95] for the full argument. 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

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

Composer does not implement this nor the sprite explicitly although nearly all image objects in the application are sprites.

Recall from Chapter 1 that the *shape* of a sprite, or any image with an alpha, is the subset of its pixels with non-0 alpha.

Some methods intended more for 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*. 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. The important point is that a subimage is not separately allocated memory. Its memory is that used by the *parent* image. 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.

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[28], 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).

[28] The word *cropped* is chosen carefully and is part of the model. The term *clipped* is reserved for geometric restriction. So boxes are cropped, but rectangles are clipped.

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—ie, 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.

An elaborate example of the use of boxes and subimages is given in [Smith90].

Image Assignment

The most fundamental image operator is image *assignment.* I will carefully discuss what it means to assign one image to another in the Altamira sprite model of image computing 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 copy.

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.

Copying, in this context, is the copying of pixels one-to-one from the source subimage defined by the intersection box to the destination subimage defined 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. Copying 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 taken care of in a box algebra computation preceding the actual assignment with the copying step. That is, in practice it is convenient to segment the general 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 routines assumes alignment and intersection have already been performed and hence, *without checking*, that the given two images are the same size.

In general, the copying of pixels must take into account that *I* and *J* might have different type and do appropriate conversions during the "copy". In the Altamira Composer implementation, we make the simplifying assumption that all images of interest have the same type across an assignment. That is, we assume that 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 assignment becomes as simple as it sounds: simple copying of *depth-aligned* channel components from one image to the other.

The generalization to arbitrary binary operator **op** is straightforward. Instead of doing a copy 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 $I = J$ **op** K.

The generalization to *n*-ary operations, for arbitrary number *n* of images, is likewise straightforward. In fact, $I = J$ **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 opera-

tion is an assignment of the result to a destination image. In the following, several of the most often used operations are explained fully: Those that composite images under the control of yet other images.

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 quadbox[4], bxoffset;
    image *Quad[4], *psubim;
    int m;
    point mask, q, c0, c1, Imaxpoint, ptoffset, ptmaxbox;
    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 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 [Smith95], the **over** operator of [PorterDuff84] 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 "*I* gets *I* minus *A* times *I* plus *J*". This routine assumes images *I*, *J*, and *A* are the same size and type, as explained above, and returns image *I*. If *I* and *J* are premultiplied images—eg, sprites—and if *A* is the alpha channel of *J*, then this routine implements *I* = *J* **over** *I*. Again, see [Smith95] for full details, including efficient programming approximations. This routine is implemented using the INT_PRELERP() macro defined there.

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*A*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. 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.

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 Chapter 1, Continuous Operators on Discrete Sprites, 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, that maps an image, reconstructed into an object with a rectangular footprint, 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 to not 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 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*[29] 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 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.

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. We 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

[29] 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 an image is used to give detail to the "surface" of a(nother) 2D image object.

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 Type 1 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*[30]. 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 apps 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. 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 rearranged 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 is 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 illus-

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

tration, programs. In a drawing program, a user works in a creative space (cf Chaper 1) 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.

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 *void* in our model. The void, or desktop, is not printed. It is not part of the creation. One unfortunate legacy of the monolithic era is the notion that images are always rectangular and that, therefore, there is always a background image "back there". In our 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 void simply doesn't print. Just as the geometrical drawing apps 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 rectangular 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[31]. 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 rectangular. 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 the illustration apps. In summary, in our model, cropping and pasting only happen at the last step, and only when instructed to happen by the user/creator.

Functions that we borrow directly from the geometrical forebears are depth ordering, alignment, and multiple selection. See any illustration program for how these work. By the way, "selection" becomes, in this model, simply pointing

[31] 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.

to a sprite—say, by clicking on it—just as in the illustration 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-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

Our model celebrates the notion that the creation of a composition and the display of it are two separate processes. 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.

In Altamira Composer, we borrowed yet another notion from the geometrical modeling world. We let 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 v 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 happen 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 [Smith90], as mentioned several times. 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 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 [Smith90] 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 void. Thus you can be sloppy in your picking if there is no ambiguity.

If you click on the void 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 a void 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.

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, stack-of-brightly-colored pebbles 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—and this is exactly what we are pursuing in the Media Foundation: 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.

REFERENCES

[PorterDuff84] Porter, Thomas, and Duff, Tom, *Compositing Digital Images*, **Computer Graphics**, Vol 18, No 3, Jul 1984, 253-259. SIGGRAPH'84 Conference Proceedings. The matting algebra and premultiplied alpha are announced to the world.

[Smith88] Smith, Alvy Ray, *Geometry and Imaging*, **Computer Graphics World**, Nov 1988, 90-94. I start drawing the fundamental distinction more sharply.

[Smith89a] Smith, Alvy Ray, *VAIL—The IceMan Volume and Imaging Language*, Tech Memo 203, Pixar, Jan 1989. The IceMan language and hence the referenced paper are proprietary to Pixar, but the objects defined in the paper are not, to the extent that they are embedded in Altamira Composer, purchased by Microsoft.

[Smith89b] Smith, Alvy Ray, *Two Useful Box Routines*, Tech Memo 216, Pixar, Dec 1989. These routines are embedded in Altamira Composer.

[Smith90] Smith, Alvy Ray, *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, Jul 1990. The concepts in this paper are embedded in Altamira Composer.

[SmithLyons92] Smith, Alvy Ray, and Lyons, Eric, *Hwb—A More Intuitive Hue-Based Color Model*, Tech Memo 0, Altamira Software Corp (also Microsoft Tech Memo 0, by inheritance), Sep 1992. This color model is embedded in Altamira Composer as its color selector method and dialog. I propose that it replace my old HSV model. It is easily superior to HSL.

[Smith95] Smith, Alvy Ray, *Image Compositing Fundamentals*, Tech Memo 4, Microsoft, Jun 1995. The argument for sprites and premultiplied alpha becomes rock solid.