Varieties of Digital Painting

Technical Memo 8

Alvy Ray Smith
August 30, 1995

Abstract

The purpose of this memo is to distinguish between the various meanings that digital painting may have. It is important to have a taxonomy so that intelligent conversation may proceed on such important issues as multi-resolution paint programs. Each type of painting will be discussed in its multi-resolution generalization. The taxonomy here splits painting into *discrete* and *continuous* categories and each of these into *maxing* and *non-maxing* subcategories.

Discrete v Continuous Painting

As usual in computer graphics, there are discrete and continuous versions of so-called *painting*, a digital technique originally intended to simulate the act of painting on canvas with a brush dipped in paint. The simplest version of painting, a discrete version, is nothing more than repeated compositing of an image called the *brush*, or *paintbrush*, with another image, often called the *background* image. The compositing (see [PorterDuff84]) repeats at every point taken from some input device such as a tablet stylus or mouse. If the brush is a single color and if the sampling of the input device is done often enough, then a stroke of paint appears to have been laid down over the background. The brush can have arbitrary shape and transparency. There are an infinity of variations of this type of painting obtained by performing an arbitrary image computation at all background pixels under the pixels of the brush in its current location. This kind of paint is the earliest realized (see [Smith78] for example) and is still used. For example, my 1993 product, Altamira Composer used this class of painting tools for its so-called “touchup” tools: paint, smear, erase, and clone, for example. It is the fortuitous overlapping of sequential brush copies that constitutes a “stroke” in this kind of painting. Absolutely *any* image can be used as a brush. Since this type of painting is defined at discrete points, using a discrete image, we shall call it *discrete painting*.

A very different class of painting is defined continuously. A geometrically defined *stroke* is used that is constrained by the hand of the artist only at selected points, called *control points*, and is rendered between these points from the model of the stroke—a cylinder with a spline backbone, for example. The model of a stroke can have transparency too. The rendered model is composited over the background image as in discrete painting. The difference is that a stroke is rendered all at once, or as a sequence of substrokes. As opposed to the discrete painting that paints only at tablet (or other input device) points, this type paints

between them—in a connect-the-dots mode—as well. This type of painting shall be called *continuous painting*. As in discrete painting, an infinity of variations is possible by using the stroke—as opposed to the brush copies—to limit those pixels of the background image to which an arbitrary image computation is performed.

There are pluses and minuses for both types of painting. I have already mentioned that arbitrary brushes may be used in the discrete case, but that a stroke is not guaranteed to be continuous. If an artist paints too fast, the copies of the paintbrush that make up the stroke can overlap only slightly, creating unsightly scallops, or can break completely apart at great speed into a string of pearls (without the string).

For the continuous case, a continuous stroke is guaranteed, by definition, but for a slow machine, the rendering of the stroke can annoyingly follow the artist's hand, in a kind of catch up mode, or for even slower machines, degenerate into unacceptably polygonal or faceted strokes due to insufficiently many interesting control points.

In the early days, I would typically include both types in a "paint program". I used the terms *painting* and *inking* to distinguish the two forms. Painting was discrete painting, because the machines were too slow to stay up with continuous rendering. I called the inking, or continuous, mode in my first paint program *sketching* ([Smith78]). One would use this tool to render a black line with antialiased edges between selected points, so it looked vaguely like pencil sketching (but it certainly didn't feel like it). It was so slow at the time that a user typically used point-by-point placement of control points rather than freehand strokes on the tablet.

On modern machines, discrete brushes can be placed so frequently as to feel like smooth stroking, without the need for any code in addition to ordinary compositing. Alternatively, they are so fast that rendering of continuous strokes can often stay up with the artist's hand. This is particularly true of continuous models such as *local* splines that can be rendered in independent pieces as the control points flood into the processor. The problem with continuous strokes is what to do at crossings, where the stroke intersects itself, and between independently rendered segments, at the overlapping endpoints. These problems are particularly visible for partially transparent strokes.

Multi-Resolution

If simple painting with a single color is the computation of interest, then multi-resolution versions of both types can be built. By definition, continuous painting is scalable and hence can be implemented at arbitrary resolution. Discrete painting is more problematic, but it too can be defined for multiple resolutions: The brush image is scaled up or down to the new resolution, using standard image resizing functionality[1], and the spacing between tablet points is cor-

[1] And translation resampling too, in general.

respondingly scaled. Then the compositing proceeds as in the original resolution case. Straightforward (simulation of) painting should look the same at all resolutions, whether realized discretely or continuously. It is the generalizations or variations on these simple schemes that are problematic.

Note that "multi-resolution" is not exactly the same as "scalable". You can think of multi-resolution as being the discrete version of scalable. A scalable feature can be applied at *any* resolution. A multi-resolution feature need only be applied at each of a given discrete set of resolutions. To make discrete painting scalable as well as *multi-res* (to coin a term to mean applicable to multiple resolutions), would require arbitrary translations of the brushes as well.

Unfortunately, multi-res discrete does not work as neatly as it may seem it should from this brief description. Let's see why not. The basic problem is that an artist's *intention* does not scale. For example, often when I am painting at the highest resolution, my intention is to actually put exactly this pixel value into that pixel—I want to paint that single red pixel on the end of his nose tan, a discrete operation. At lower resolution, this is a meaningless operation: In the lower resolutions, that single miscreant pixel—from a bad scan, say—is not likely to be visible even.

Another way to state the basic problem in the multi-res discrete case is that many functions implemented under the rubric of painting are inherently resolution dependent. Recall that the general meaning of painting is any image computation under hand control—equivalently, under paintbrush control. So for an extreme example, let's invent "flicker paint". On interlaced televisions, a flicker phenomenon is introduced by coloring alternate scanlines in highly contrasting colors—eg, odd lines are black and even ones white. A multi-res realization of flicker paint would be effective (or intentional, again) only at full resolution. Any lower resolution representation would be mush—gray in our example.

It is worth noting, however, that the problems of intention as represented by the two examples above can be solved by requiring that a user paints at the highest resolution. So the paint program would require that a user work on an area to be painted at the "truth" level of resolution. Alternatively, the user paints at the displayed resolution and experiences odd results at higher resolutions. For example, if he flicker paints at screen resolution so that every other screen scanline is painted black or white, then at higher resolution he gets "fat" scanlines—eg at four times screen res he would get four white lines alternating with four black ones under the scaled (and possibly translated) brush.

It is also worth noting that most ordinary kinds of discrete painting should work at multi-res. We discuss several examples below.

Examples of Discrete Painting

The normal discrete varieties implemented in Altamira Composer are Paint Over and Paint Atop[2] for simple painting with a single color, Smear Over and Smear Atop for smearing in the direction of brush motion, Erase for erasing holes in a sprite (ie, in its alpha channel), Dodge/Burn for local darkening and brightening, Step Contrast for local contrast changing, Xfer and Clone for relative and absolute cloning[3], Tint and Colorize for local colorization[4]. All of these should scale in a multi-res realization by the technique outlined above: Scale the brush, scale the location, apply the function to the new res sprite.

It is user feedback that is the critical issue. Any realization of painting must be fast enough that the user is convinced he is painting on a sprite. It does not matter how the implementation is done, so long as the illusion is not broken. Smearing, for example, is often used to "mix paints" together in subtle ways, as on a real artist's palette. It is very important that the user immediately see the results of "schmudging" the paints together so as not to go "too far". A realization that uses intermediate buffers between the paint and the sprite that is apparently being modified (it need not be) must preserve the illusion although rather numerous copying and modifying operations of pixels between the sprite and buffer (could be another sprite) may be occurring in the realization. Another form of intermediate buffering, called maxing, is discussed in a following section.

A non-standard type of discrete painting in Altamira Composer is Impression painting, named for a vague resemblance in effect to impressionistic painting. The algorithm here is this: Determine the color of the pixel under the center of the brush, change the paint color on the brush to this color, and lay down (composite over) one copy of the brush, then repeat for the next brush location. This tends to "de-res" (lower the resolution) the image in a round, shower-door, kind of way that can be quite interesting. This would give different results at lower resolution than at full resolution—because the color of the center pixel, assuming that can always be well defined, is generally an average of several colors in the true image. But again, the problem disappears if Impression painting is allowed only at full resolution.

---

[2] Over and atop refer to the matting algebra operators of [PorterDuff84]. Paint Atop a sprite, for example, applies paint to the non-transparent pixels of the sprite only, Paint Over to all pixels in the bounding box of the sprite.

[3] Xfer and Clone are both pixel transfer operations under brush control. If the transfer is from an absolute, fixed location, it is called cloning, or rubberstamping—a very dumb operation. Relative transfers are from a source location that is relative the target location, indicated by the user, by a fixed vector. This transfer an area of pixels from one place to another and is usually much more useful than dumb cloning. Cloning unfortunately is often used to have both meanings, which makes conversaton difficult.

[4] Tint changes the hue of a picture but preserves its whites and blacks. Colorize changes the hue of whites and blacks as well, as if a colored filter were placed over the image where the brush passes.

Another class of non-standard discrete painting types in Altamira Composer are the Warp paints. Several of the Warp operations available in the program can be applied under hand, or brush, control. For example, Vortex paint causes the image under the brush to be deformed as if it were being twisted about its center point. All of these functions are defined with continuous deformations, realized with discrete resampling operations. They should all scale to multiple resolutions. These are offered here to bolster the definition of painting to be any image computation under hand control. The other Warp paints are Bulge, Escher, Mesa, Radial Sweep, and Spoke Inversion.

Read-Modify-Write Painting

Smearing is a type of discrete painting that deserves further explication. It will lead us to draw another distinction. The way smearing works in Altamira Composer is this: The contents of the current sprite in the pixels under the brush are copied into the brush, translated one pixel in the direction of motion (eight directions honored), and lerped[5] back into the current sprite. The lerp is weighted by the current opacity. This is repeated again and again as the brush is moved across the sprite. The effect is that portions of the image are moved, or smeared, in the direction of motion, with transparency serving as viscosity (the lower the opacity slider, the more viscous is the apparent smearing). So this type of painting does a read-modify-write cycle at every brush position. Another way to say this is that there is feedback between the sprite and the brush; the results of the previous brush applications affect the current application. It is clear that there are many variations possible using this paradigm. For example, the modification could be a blur or sharpen rather than a shift.

A related implementation would maintain a copy of the source sprite, before any smearing. Then the first step at each new position of the brush is to copy pixels into the brush from the source sprite, not the sprite under modification. Any painting function implemented in the (read modified image)-modify-write form can also be implemented in (read original image)-modify-write form, but the results will generally be different in the two cases. Let's call the two cases *modified RMW* and *original RMW* for short. The blur/sharpen example above in fact does not give the intended result in the former case (modified RMW) but does in the latter. In the former case, it tends to degenerate the image while in the latter, it sharpens the image in the local areas defined by hand.

This distinction affects the multi-res generalization. In the modified RMW case, the order of brush positions is always important so they have to be remembered for the multi-res case. There can be thousands of these positions, which is not unthinkable but does lead to management problems. In the original RMW case, the order is often not important. It is important sometimes, however, For example, the smearing algorithm described above implemented in original RMW form gives a different result, in general, if the order of translation of brush con-

[5] Lerp is short for linearly interpolated.

tents is altered. But if the brush contents aren't moved during the RMW cycle, then the order of brush application is immaterial. For a multi-res implementation in these cases then, the brush positions do not have to be remembered from one resolution to another. Often just a representation of "dirty pixels" or some other "scalar field" need be remembered for the multi-res application, and these can be quite succinct.

Examples of Continuous Painting

There is an example of continuous painting in Altamira Composer too. This is buried in the Spline/Polygon package. If the Open/Not Tapered or the Closed/Not Filled options are selected then the splines (or polygons) are rendered with shape. That is, the continuous stroke defined by the spline is rendered with a constant width and antialiased edges and carefully rounded endpoints (in the Open case). This is not implemented as painting in Altamira Composer, but it could be. Instead, the knot, or duck, points of the spline are appended one at a time and then the spline is rendered when the list of ducks is complete. To make this a painting operation would require the rendering of the spline as the ducks were being sampled from a continuous mouse movement. This is possible because the Duff Splines used in Altamira Composer are of the local variety (as opposed to global). This means that early parts of the spline are not affected by later parts, so can be rendered as defined. The rendering algorithm must ensure that there are no artifacts at the joints between successively rendered segments of the spline. With these conditions in place it should be straightforward to scale the spline stroking operation—ie, continuous painting.

For Open/Tapered splines, color, opacity, and width can vary continuously along the spline stroke. However, this cannot be implemented until all the ducks are known, so that the interpolation length can be determined. It is not clear how this could be made comfortably interactive. We did implement it at Lucasfilm with a virtual stroke with the mouse (tablet stylus actually), followed by the rendering. One had to guess where the strokes would fall, but it lead to beautifully graceful strokes.

Waxy Buildup

Simulation of airbrushing[6] can be accomplished in both ways, discretely and continuously. The simple way is by lerping a nicely shaped brush in a constant color with a background image. The complex way is to simulate a stroke of airbrush paint with a geometric, or continuous, model - complete in the most elaborate cases with simulation of the tilt of the airbrush flow. In both methods, you may want to do what we call "max paint" in Altamira Composer - that is, you may want to have the paint build up to a maximum opacity but not exceed it.

[6] For years I fought a losing battle to restrict the use of the term "airbrushing" to a true simulation of the airbrush which builds up a surface with randomly sprayed particles of pigment. I have now given up and use the common computer graphics meaning of a very soft-edged lerp of a continuous, or apparently continuous, surface to an image.

One way to accomplish this is to paint into an empty buffer where you can check each pixel as it is about to be written to determine if it "is full" (to the preset opacity) already. If it is, then no further modification is performed. Then when the stroke is completely rendered, it is composited into the background. So this could more generally be called “check paint” to directly imply all forms of checking against “dirty” pixels. So this is related to the scalar field discussion above.

Summary of Painting Types

We have the following kinds of painting classes:

- **Discrete, non-maxing**. Each brush copy is repositioned to a position sampled from the user input and is combined directly with the background independently of any other copy of the brush. *Any* imaging operation can be performed under the brush here, not just compositing. This works well if there is sufficient processor speed to sample user input very often. In some cases, airbrushing for example, it can cause undesired buildup of opacity.
- **Discrete, maxing**. Each brush copy is repositioned to a position sampled from the user input and is composited into a temporary image buffer before compositing with the background. This is so each pixel can be checked for "maxing", where it is understood that maxing is just one representative check that can be made at this step. The buffer starts empty. This can be made to work quite well but uses more memory. It avoids the "waxy buildup" problem. The extra memory can be used for some nice editing (or undo) features and can also be used in some cases to avoid remembering brush positions.
- **Continuous, non-maxing**. A stroke is modeled from a model of a brush. For example, a gaussian brush model is convolved with a spline through user input points. The model is rendered directly into the background image. This works well for processes that can be continuously modeled. It can suffer from the waxy buildup problem at joints between rendered segments or when strokes cross over themselves. On systems with insufficient speed, there can be a very annoying delay between a user's movement and the rendering of the stroke. Bad implementations don't understand that strokes of length 0 should also be rendered (for spots).
- **Continuous, maxing**. A continuous model is rendered into a temporary buffer so that the maxing check can be made.

As already mention, Altamira Composer has examples of the first three types. Most so-called painting or touchup functions are of the two discrete types depending on the desired function. The continuous, non-maxing type is hidden in the spline rendering package which is missing only a freeform user input to complete. The current implementation does not allow freeform input - the user enters the control points, then says go. This would be simple to change but was not the intent of Composer. And strictly speaking, maxing is partially implemented: It is implemented along the spline at its joints but not at other self-intersections (to save memory). This seems to be a decent compromise.

Then, of course, there are mixtures of discrete and continuous. For example, a spline could be defined and then discrete brushes laid down along it at discrete intervals. Other variations are possible.

A Patent Distinction

Then there is another distinction that can be made—that enters in patent suits: When rendering a continuous model, you can do it two ways in the case of an unchanging brush shape moving along a geometric curve:

- Render copies of the brush along the curve.
- Compute a single continuous shape from the brush shape and the curve shape and render this.

The first rendering type is what Quantel has patented (although they claim everything and all ways)[7]. If you do the math, you can show that with appropriate spacing of the brush copies, both methods generate the same result. A bunch of gaussian spots rendered atop each other along a straight line, for example, is the same as the rendering of a cylinder with gaussian cross section, that lies along the line (with appropriate treatment of the endpoints).

Conclusions

The first observation is that the concept of "painting" in imaging is quite a bit more complex than a simple mention of the term often implies. For example, the comment that prompted this memo was roughly, "Painting should easily be realizable in a multi-resolution format, shouldn't it?" It is not obvious, considering all the different things "painting" means, that there is a simple Yes or No answer to this question.

But now, after careful analysis of all the different types of painting, I believe it is safe to say that all types considered here *are* capable of multi-resolution implementation, though certainly not with a single scheme (cf, modified v original read-modify-write painting types and order dependent v order independent types) and being mindful that certain operations just don't make sense other than at a single resolution (cf, flicker paint) and that others give different results at different resolutions (cf, Impression paint). A further caveat is that an insufficiently fast realization is not good enough even though it might "work" programmatically.

A large class of painting effects that is not analyzed here are the so-called "painterly effects" that often attempt to simulate well-known "real" media affects such as brush hairs, directional painting, chalk, etc (eg, see [Strassmann86]). These would probably have to be analyzed on a case-by-case basis. I believe that some of the concepts described here might be useful for such further analysis.

[7] Many of us in the industry believe these patents should be struck down. They have not yet successfully been so, however. [Note added 16 Dec 1998: They now have been struck down.]

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 classic digital compositing paper.

[Smith78] Smith, Alvy Ray, *Paint*, Tech Memo 7, Computer Graphics Lab, New York Institute of Technology, Old Westbury, NY, Jul 1978. Issued as tutorial notes at SIGGRAPHs 78-82. Reprinted in **Tutorial: Computer Graphics**, edited by John C Beatty and Kellogg S Booth, IEEE Computer Society Press, Silver Spring, Maryland, 2nd edition, 1982, 501-515. Contains a brief history of paint. First public mention of RGB paint, airbrushing, etc.

[Strassmann86] Strassmann, Steve, *Hairy Brushes*, **Computer Graphics**, Vol 20, No 4, Aug 1986, 225-232. SIGGRAPH'86 Conference Proceedings.

[Whitted83] Whitted, Turner, *Anti-Aliased Line Drawing Using Brush Extrusion*, **Computer Graphics**, Vol 17, No 3, Jul 1983, 151-156. SIGGRAPH'83 Conference Proceedings.