Live Paint:

Painting with Procedural Multiscale Textures

Ken Perlin
Media Research Laboratory
Courant Institute of Mathematical Sciences – New York University

Luiz Velho
IMPA – Instituto de Matemática Pura e Aplicada

Abstract

We present actively procedural multiresolution paint textures. Texture elements may be linearly combined to create complex composite textures that continue to refine themselves when viewed at successively greater magnification. Actively procedural textures constitute a powerful drawing tool that can be used in a multiresolution paint system. They provide a mechanism to generate an infinite amount of detail with a simple and compact representation. We give several examples of procedural textures and show how to create different painting effects with them.

1 Introduction

The introduction of multiresolution paint systems is a recent development in the field of computer graphics, [10], [1], [5], [12]. In this type of system, the user can view and modify an image at any desired resolution. This is possible because the internal image representation supports multiple levels of detail.

In multiresolution paint systems it is possible to make modifications at different image magnifications. The user can quickly make coarse changes over large areas of the picture, as well as fine and precise changes over small areas. Although this capability provides a great degree of control over the painting process, it is the painting tool that ultimately determines what goes into the picture. For this reason, it is necessary to develop tools that can take full advantage of multiresolution paint systems.

Standard painting tools are designed to operate at fixed resolution and, therefore, can generate only a certain amount of image detail, commensurate with resolution. When the user wants to create fine detail over a large area of the image, s/he must paint at a high magnification level. This task is time consuming, even if a multiresolution paint system is used.

The power of multiresolution paint systems is enhanced if painting tools are able to exploit the underlying multiresolution image representation by operating at multiple levels of detail. In this way, when the user wants to paint a complex pattern over a large area of the picture, s/he can perform the operation at a coarse level very quickly and still create as much detail as desired.

In this paper, we develop a framework for the design and implementation of multiresolution painting tools. This framework is based on general procedural textures defined over both spatial and scale domains. These procedures are executed at multiple resolution levels in order to add texture details to the picture. Basic texture elements can be linearly combined to create complex composite textured brushes. This framework fits naturally into the context of multiresolution paint systems.

The structure of this paper is as follows: Section 2 reviews the principles of multiresolution paint systems; Section 3 gives a general overview of multiresolution textures; Section 4 shows where texture is invoked in the system; Section 5 explains how the texture refinement method works; Section 6 illustrates the system in action with examples and pretty pictures; Section 7 discusses the use of images in our framework; and Section 8 concludes with final remarks and a discussion of future work.

2 Multiresolution Paint Systems

A multiresolution paint system allows the user to view and modify an image at multiple resolution levels.

The painting process consists of a cycle in which the following tasks are repeatedly executed:

$$\begin{array}{ccc} & \text{modify image at level } x & \\ \downarrow & & \uparrow \\ & \text{move up or down a level} & \end{array}$$

For this, the system must support:

- Multiresolution Image Representation;
- Painting and Compositing at Multiple Levels.

We can classify multiresolution paint systems with respect to the way they implement the image data structure and the operations mentioned above.

2.1 Multiresolution Image Representations

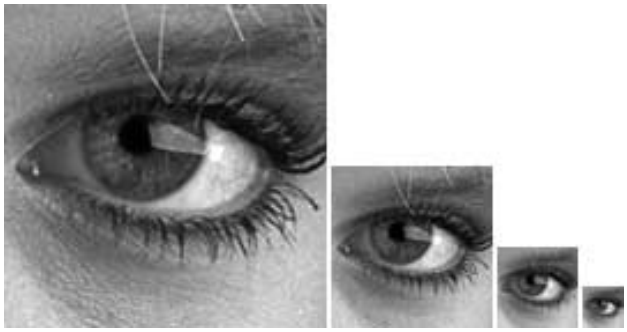
There are two basic ways to represent an image at multiple resolutions: with a *lowpass* pyramid or with a *bandpass* pyramid.

[1] Courant Institute of Mathematical Sciences, New York University, 719 Broadway 12th Floor, New York, NY, 10003. **perlin@cs.nyu.edu**

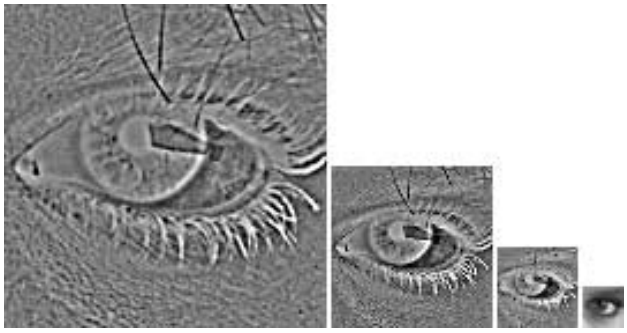
[2] IMPA – Instituto de Matemática Pura e Aplicada, Estrada Dona Castorina 110, Rio de Janeiro, RJ, Brazil, 22460-320. **lvelho@visgraf.impa.br**

A lowpass pyramid consists of multiple copies of the image at different resolutions, [11]. In Figure 1(a) we show an example of a lowpass pyramid with 4 levels.

A bandpass pyramid consists of a coarse resolution version of the image, together with a sequence of image details that are required to produce finer resolution versions of the image from coarse versions, [2]. The bandpass pyramid can be constructed by taking differences between consecutive levels of the lowpass pyramid. In Figure 1(b) we show an example of a bandpass pyramid with 4 levels. The images in the bandpass pyramid may contain negative values. For this reason, the figure shows zero values as middle gray.



(a)



(b)

Figure 1: Lowpass (a) and Bandpass (b) Pyramids

Note that the lowpass pyramid is a redundant representation since the same information is present at all levels of the pyramid. The bandpass pyramid, on the other hand, is not redundant because it keeps only the information necessary to go from coarse to fine resolution levels.

A Wavelet pyramid is a kind of bandpass pyramid which discriminates image details along the horizontal, vertical and diagonal directions, [6].

2.2 Multiresolution Painting and Compositing

There are two main ways in which image operations can be incorporated into a multiresolution paint system: by re-execution or by lazy evaluation.

If the system allows the image to be represented at arbitrarily high resolutions, then it is not possible to perform image operations at all levels simultaneously during the interaction cycle. Multiresolution paint systems generally deal with this as follows: while the user is painting, the system only updates the currently visible level. Changes to the image are cached, to be propagated to other resolution levels at a later time. What distinguishes different implementations of image operations is the strategy used to postpone the propagation of changes.

If a re-execution strategy is used, changes are cached as prescriptions for redrawing. When the user moves to another level of detail, all cached operations are re-executed at that level.

If a lazy evaluation strategy is used, modifications to the image are iteratively propagated through the image pyramid when the user magnifies the view to successively more detailed levels. Changes that will affect higher resolutions are evaluated only when the user first magnifies the view sufficiently to see these resolutions.

2.3 Classification of Multiresolution Paint Systems

The multiresolution image data structures and operations described above have a direct correspondence with one another. Accordingly, there are two types of multiresolution paint systems:

I – Lowpass pyramid + Re-execution strategy

II – Bandpass pyramid + Lazy evaluation strategy

Examples of type *I* systems are Live Picture [5] and XRes [12]. Examples of type *II* systems are the Haar wavelet [1] and the B-spline wavelet [10] paint programs.

The general idea of multiresolution painting tools applies equally well to painting systems of types *I* and *II*. Although in this paper we will emphasize implementation techniques that are more suited to systems of type *II*, the same techniques could be used with proper changes in systems of type *I*.

3 Multiresolution Textures

In this section we introduce the concept of multiscale textures and discuss how they are implemented.

3.1 Motivation

Let us say that a user of a paint system wishes to paint with a "rock" generating brush. After painting, the user should subsequently be able to zoom in and continue to see ever finer details of the rock texture.

Alternatively, if the user paints with a "checkerboard" brush, then no matter how far s/he zooms into the checkerboard texture, the boundaries between the white and black squares should always remain sharp.

Let us now suppose that the user paints some rock, then zooms in a bit and paints some translucent checkerboard over the rock. As the user zooms in, arbitrarily small rock details should still be visible behind the checkerboard, but attenuated. Both of the composited textures should continue to reveal more high frequency detail as the user's view zooms in.

The user should, for example, be able to paint with a brush which consists of one part checkerboard and two parts rock, or in fact to freely mix any such texture elements.

In this manner, the user should be able to paint with and to composite many different layers of procedural texture, with the expectation that all visible layers will appear in the proper proportion at all levels of detail.

3.2 How to do this

In order to implement this behavior we have developed a model for multiscale painted texture that allows procedural textures to be combined additively.

The key insight is that texture must be added in two distinct ways:

(1) Whenever the user paints a texture, the system must display that texture's initial appearance.

(2) As the user successively magnifies the view, the system must continue to add detail to the painted texture.

It is the responsibility of the texture procedure to perform these two functions.

To make this all happen properly and efficiently, we need two tools: Procedural Bandpass Pyramids and Procedural Ink.

3.3 Procedural Bandpass Pyramids

To do sharpening properly, we use procedural bandpass pyramids. First let us briefly review bandpass image pyramids. Each level of a bandpass pyramid gives the difference in detail between successive submagnifications of an image. Consider an image of resolution $2^n \times 2^n$. If we want to view this image at a resolution of $2^{n-1} \times 2^{n-1}$, we can blur it using a smoothing filter and then decimate. If the image is then blown up again with a good interpolation filter, it will appear blurry. Information has been lost. We must add a correction to each pixel of this blurry image in order to recreate the original image. This correction is itself a $2^n \times 2^n$ image containing the image details that were lost.

If we apply the same process to the decimated image, recursively, we can create a sequence of such detail images with descending resolutions: $2^n \times 2^n$, $2^{n-1} \times 2^{n-1}$, $\ldots 2 \times 2$. This sequence of images constitutes a bandpass pyramid [2].

Once we have its bandpass pyramid, we can reconstitute an image at any submagnification. Beginning with a single pixel, we magnify and add the 2×2 bandpass image, then repeat with the 4×4 bandpass image, and so on. The pixels of the $2^k \times 2^k$ image of a bandpass pyramid are called "level-k bandpass coefficients".

A multiresolution paint system can show an image at various scales. Bandpass coefficients make up the difference between the less detailed view that is visible when the image is small, and the more detailed image that is available after magnifying by two. If we look at it this way, we can see that as we continue to magnify the user's view of a texture, our texture procedure should correspondingly add in the next level of bandpass coefficients, so that the texture will always have sharp detail.

Images are finite. At some point a multiresolution paint system will magnify the view beyond any stored image's resolution. For levels beyond this, the image simply becomes blurrier, since it can contribute no more bandpass coefficients. In other words, the image's bandpass pyramid is of finite depth.

But procedural textures are not so restricted. We can define a "procedural" bandpass pyramid which given a type, and values for x, y, and *level*, returns the difference in a texture's appearance when viewed at successive magnifications. Procedural bandpass pyramids can be of infinite depth.

Here is a simple example. It is well known that the appearance of rock can be synthesized by $1/f$ noise [9]. The difference in this texture's appearance between successive magnifications is just the addition of attenuated random noise at the higher magnification. A procedure that simulates a bandpass pyramid with this behavior is quite simple to define:

$$\text{add\_texture}(\text{ROCK}, x, y, \textit{level}) := \frac{\text{pseudorandom}(x, y, \textit{level})}{2^{\textit{level}/2}}$$

where the pseudorandom function is implemented by the same permutation method used to choose pseudorandom gradients from Z^3 for the Noise function [9].

The tricky aspect of this process is that we need to sharpen each painted texture only the first time that the view is magnified to a new level of greatest detail. For example, if the user paints a texture, then zooms in and out a few times, we do not want to add bandpass coefficients to the texture again and again. The result would be incorrect. For this reason, texture propagation is controlled by procedural "ink".

3.4 Procedural Ink

In the sections that follow, we will use the phrase "texture instance" to refer to a primitive texture component. Some examples are: rock of a particular scale, a checkerboard of a certain size, sawtooth stripes of a particular width, or a source image texture painted on at a certain scale. We build all textures as combinations of texture instances.

Procedural ink is texture in latent state. We represent it as a vector of amplitudes; each element of the ink vector modulates the amplitude of one texture instance. The first ink channel is reserved for *ink.alpha* – the attenuation of the ink vector itself. All elements of the ink vector will be attenuated by this *ink.alpha*.

Textures are mixed by compositing and layering ink vectors. The resulting composite ink vector is then used to control a texture generator procedure. Here is a key point: Because all elements of an ink vector are premultiplied by *ink.alpha*, we can mix textures simply by adding various ink vectors.

As noted above, we want to do sharpening only the first time that a viewer magnifies a painted area to a new view. In order to accomplish this we need a form of lazy evaluation. This is where the ink comes in. Ink controls when the texture gets sharpened. The key property of ink is that it flows downhill – toward levels of ever greater detail. As ink flows down, it causes the system to refine those texture instances which the ink modulates.

More precisely, each element of the ink vector is used to scale the bandpass coefficients that must be added in order to sharpen one texture instance.

Ink is only used for texture refinement at the moment that it first enters a view level. This will happen only in one case: when there is ink at the current view, and then the user magnifies the view. At this time, the used ink leaves the coarser level, and pools at the more detailed level, waiting for the user to further magnify the view.

Note that there can be considerable delay between the time that a texture is painted and the time that this lazy-evaluation sharpening occurs. For example, the user might paint at some level, then *decrease* the magnification, zooming out to paint something somewhere else. Meanwhile, the ink is still sitting at the former level. When the user later returns to that view and magnifies, only then will the ink continue its downward journey.

4 Where Texture is called

Texture is called in two places: once as the user interactively paints, and again when the view is magnified, in order to add bandpass coefficients.

4.1 Texturing during painting

When the user paints at a sample using *drawing_ink*, and with opacity *drawing_alpha*, then the *color*, *alpha* and *ink* at the sample are modified as follows:

(1) *color* := texture(*drawing_ink*) $\text{OVER}_{\textit{drawing\_alpha}}$ *color*

(2) *alpha* := 1.0 $\text{OVER}_{\textit{drawing\_alpha}}$ *alpha*

(3) *ink* := *drawing_ink* $\text{OVER}_{\textit{drawing\_alpha}}$ *ink*

where "b OVER_t a" denotes linear interpolation $a + t(b - a)$, *texture* is a compound procedural texture as defined in subsection 5.2, and *ink* is a prescription for evaluating the procedural texture.

This is the point where the ink is first injected into the system, and where the first, coarse view of the texture is painted. Note that the new color is merged directly into the sample. There is no need to explicitly store back-to-front layers.

4.2 Texture refinement during magnification

Texture is also invoked when the user increases the magnification level of the view. As the user's view changes from a coarser level to a more detailed level, ink flows down to this new level, so that more texture detail can be induced.

When the view is magnified from a coarser to a more detailed level, we need to propagate both *color* and *ink* to the new level. To describe this process, we define:

- *detail*: the portion of a sample's color at the more detailed level that was too finely detailed to be visible at the coarser level. This quantity is generated by the multiresolution paint system at the moment that magnification was reduced. If this is the first time that we have ever visited the more detailed level, then this quantity will be zero. In our implementation, this quantity is computed from B-spline wavelets.
- *coarse*: a magnified view of what is currently visible at the coarser level.
- *new_ink*: a magnified view of the new ink from all coarser levels which now needs to flow down to this more detailed level.

Let us first review the magnification procedure in a multiresolution paint system that uses lazy evaluation, but that does not support procedural texture. In such a system, *ink* is just a vector of [*red*, *green*, *blue*, *alpha*].

Let us consider the situation where the user has painted with *new_ink* at some coarser level *after* the last time s/he had visited the next more detailed level. Now the user wants to magnify the view. Because the system has lazy evaluation, this *new_ink* will not yet be incorporated into the more detailed level. To incorporate this *new_ink*, we do the following steps at the more detailed level:

(1) *color* := *coarse* + (1 - *new_ink.alpha*) * *detail*

(2) *ink* := *new_ink* $\text{OVER}_{new\_ink.alpha}$ *ink*

Notice that we do not do an OVER operation in step (1). This is because the *coarse* value already incorporates all *new_ink* that has been painted at all coarser levels. Only the detail is out of date.

After this, we erase (i.e. set to zero) all the ink at the coarse level that has flowed to the more detailed level. The general effect is that as the user's view is progressively magnified, *ink* at coarser levels continually "flows down" to more detailed levels. As it does so, its appearance becomes progressively blurrier.

To support procedural texture, we need only to add a term to step (1):

(1) *color* := *coarse* + texture(*new_ink*)
+ (1 - *new_ink.alpha*) * *detail*

Note that all elements of *new_ink* are already attenuated by *new_ink.alpha*. The result is that all added texture is attenuated by *new_ink.alpha*. That is why we simply add texture, instead of overlaying it on top of the detail.

Now as the ink flows down, it is continually refined.

5 From Ink to Texture

Texture comes in different "types". Each type requires a different refinement method.

In the following sections, we describe how texture is combined, show various texture procedures, and give some examples.

5.1 Types and Instances

As described above, a procedural texture is built up by adding bandpass coefficients to each texture instance modulated by the ink vector, at successive levels of detail.

The "type" of a texture instance identifies the method that is used for adding bandpass coefficients at each level. The method chosen will determine the general look of the texture.

Each texture instance is identified by:

- its type
- its base magnification level (equal to the view level at which the user painted the texture).

A unique element of the ink vector is allocated to each texture instance, the first time that instance is painted. This ink element contains the amplitude that will be used to scale the bandpass coefficients as that texture instance is refined.

5.2 Adding Texture

All texture instances are combined linearly. The texture procedure simply loops through the ink vector and sums the contribution from each instance i:

$$\sum_i amplitude_i \ * \ \text{add\_texture}(type_i, x, y, level - base\_level_i)$$

Note that this arrangement allows us to linearly combine different ink vectors. As any ink element is attenuated, the detail values added to its corresponding texture instance will be equally attenuated.

5.3 Some Examples of Texture Types

For each type of texture, there is a corresponding procedural bandpass pyramid; a function that computes how much detail must be added into that texture at each level. We now describe the procedural bandpass pyramid used to construct various specific types of texture. Each type is specified in **Boldface**, followed by its refinement method.

The variable *dlevel* below refers to the difference between the current level and the texture instance's base magnification level. Note that *dlevel* = 0 when the first coarse texture is painted, and that *dlevel* > 0 whenever the texture is subsequently sharpened.

White:
if *dlevel* = 0 then 1 else 0

Rock:
pseudorandom(x,y,*dlevel*) / $2^{dlevel/2}$

Stripes:
sqr_wave(x) / $(dlevel + 0.4)^{0.1}$

Sawtooth:
saw_wave(x) / $(dlevel + 1)^{0.14}$

Squares:
sqr_wave(x) * sqr_wave(y) / $(dlevel + 0.1)^{0.14}$

where sqr_wave and saw_wave are defined as follows:

```
square_wave(x)
    if i = 0 or i = n/2-1 then 1
    else if i = n/2 or i = n-1 then -1 else 0

saw_wave(x)
    if i = 0 then 1 else if i = n-1 then -1 else 0
```

with n := $2^{dlevel+1}$ and i := $x \bmod n$.

In practice each of the above power curves is computed only once, and then stored in a table, which is subsequently indexed by *dlevel*. These particular power curves depend on the B-spline reconstruction kernel that we use, [10]. A system with a different reconstruction kernel would require different curves.

White is handled as a special case, since White texture requires no sharpening. For this reason, in practice we modulate all White texture instances in one element of *drawing_ink*. We change the amplitude of this element each time the user lightens or darkens the brush.

6 Examples

In this section we give some examples of the procedural multiresolution textures and their use. First we show some examples of magnifying and compositing textures. Then we present a more advanced example: creating an entire multiresolution terrain model by painting with procedural textures.

6.1 Simple Examples

Figure 2(a) shows the name of a beautiful city written with a sawtooth generating brush. Figure 2(b) is a magnified view into the dot above the letter "i". Figure 2(c) is magnified even further. Each image is four times the magnification of the previous image. We note that the intensity ramps in the horizontal direction are piecewise linear.

Figure 3 shows successive magnifications of a translucent blending of three active textures: rock, squares, and horizontal stripes. Note how in Figure 3(c) the squares texture smoothly blends into the stripes texture.

Figure 4 shows the effect of a smoothing brush. This brush simply averages neighboring values in the image, and blends the result with White ink. The White ink element modulates brightness; the alpha ink element modulates opacity. The most important effect of this brush is that it locally reduces or eliminates texture sharpening. Figure 4(a) shows rock texture. Figure 4(b) shows an “X” drawn over this with a smoothing brush. Figure 4(c) is a magnified view into just below the central cross of the “X”.

6.2 Advanced Example: A Terrain Model

Terrain modeling is an important application where fractal images may be used to describe elevation data [3], [4]. This example exploits the expressiveness of procedural textures in a system that combines interactive painting with 3D visualization.

Figure 5 shows a terrain being interactively modeled in the system. On the right, the user sees an interactive height field view of the intensity image. First we will describe this interaction tool, and then we will discuss the example in Figure 5.

The user sees a perspective view of intensity, as a height field mesh. The mesh is rendered back to front, so no Z buffer is required. The brightness at each location on this mesh is composed of a weighted sum of that location’s height and directional derivatives. The user can interactively pan and tilt this view with the mouse.

In order to maintain interactivity on platforms that do not have polygon transformation hardware, we use progressive refinement. The user can change the view and see the results in real time over a coarse mesh approximation. Then while the user goes back to painting, the terrain model gradually increases to full resolution over several seconds, as a background activity. Any further changes to the view during this time will interrupt the refinement process and start it again.

In figure 5(a) we see a height field created by magnifying the view into a squiggle drawn with a rock texture. In figure 5(b) we have drawn a new rock texture instance as a wavy horizontal across the image. In the height field view, this appears as a mountainous ridge across the terrain. We also have drawn with a dark erasing brush near the bottom of the image to simulate the flat terrain of a river.

In figure 5(c) we magnify the view into the bay that appeared in the lower left of figure 5(b). Then, we sprinkle some individual bright squares near the river. In the height field view these appear as tall buildings. Note that the edges of these buildings will be perfectly sharp, no matter how close we get.

In this same figure, we have also used a transparent brush to paint some squares generating texture at several scales, in order to simulate a large cityscape. This can be seen at the left edge of the image. As we paint, the opacity of the brush controls the height of the buildings at the brush. The more time we spend over any area, the taller the buildings grow.

This entire process took less than a minute of painting.

7 Using Images

In this section we discuss how images can be used to complement our framework for multiresolution procedural textures.

7.1 Images as Procedural Brushes

In addition to using textures that are strictly procedural, we can use a multiresolution image as a texture source [7]. This enables many useful paint operations and provides a way to seamlessly incorporate multiresolution images into the system.

A multiresolution source image is represented internally by a bandpass pyramid, ie. a coarse image at the base resolution level and a sequence of detail images for the other levels.

The corresponding data structure is:

- size of the base image ($n \times m$)
- number of levels of the bandpass pyramid
- bandpass pyramid

The encapsulation of images in a procedural brush is done by associating a source image with a texture instance. When the user selects a source image to paint, a new texture is instantiated: an index of the ink vector is allocated, a reference from this index to the data structure representing the selected image is established, and the base level is set to the current resolution level.

An image texture is similar to a procedural texture with the exception that the values are copied from its bandpass pyramid instead of being generated algorithmically. The operation is divided in two parts: at the base level pixels are copied from the base image onto the canvas; when the user changes level and the image needs to be refined, the bandpass coefficients are added.

Before using a source image as a procedural brush, we need first to construct a bandpass pyramid for it. For this purpose, we employ a modified version of the wavelet transform engine used in the system. This can be done as an independent operation by a program that builds a library of source image brushes or as a built-in operation of the paint system to create a source image brush from any rectangular region of the canvas.

Source images can be replicated by the procedural brush to cover a larger region of the canvas by using a rectangular tiling arrangement. Image replication has to be taken into account in the construction of the bandpass pyramid. If we wish the texture to tile the source image, then we must employ “toroidal” end conditions when building the pyramid, with both horizontal and vertical wraparound.

7.2 Synthesizing Textures from Image Samples

The design of procedural textures usually requires some kind of programming [9]. A powerful alternative to this way of creating procedural textures is the automatic generation of a procedural bandpass pyramid from a sample image of the texture.

For this, we need a mechanism to predict the coefficients of the next level of the pyramid based on the current ones. In some cases, the prediction rules are very simple. For example, in the case of perfectly sharp edges and fractals the coefficients are multiples of each other as shown in subsection 5.3.

In the general case, the mappings between coefficients at different levels are inherently non-linear and multi-modal. We are currently investigating a technique to generate these mappings from a statistical analysis of a sample texture. This technique is similar to the one used for image compression in [8]. The method is based on a vector quantization of the bandpass pyramid and a subsequent analysis of the correlation between the codes generated by the vector quantizer. This analysis allows us to build a prediction table for each code in a codebook, which gives a mapping from coarser to more detailed bandpass coefficients. The texture procedure then uses the prediction tables to generate new bandpass coefficients during refinement.

8 Conclusions

In this paper we have introduced the concept of actively procedural multiresolution paint textures. These are live picture elements that continue to refine themselves when viewed at magnifications greater than the one at which they were originally painted.

8.1 Summary

We presented a framework to implement these active textures and incorporate them as procedural brushes into a multiresolution paint system.

We have described a simple way to linearly combine primitive texture elements in order to create complex composite textures. We have explained how to use source images in procedural brushes. We have discussed how to automatically generate multiresolution procedural textures from an image sample of the texture. We have given several examples of the use of our framework in a multiresolution paint system.

In conclusion, active procedural texture constitutes a powerful new painting tool to more fully exploit the power of multiresolution paint systems.

8.2 Future Work

Future work should go in several main directions:

- Enhancing the capabilities of the current framework;
- Adding more texture generators of interest;
- Developing an environment for the design of procedural textures;
- Investigating extrapolation techniques to generate missing image details.

The current framework could be enhanced by incorporating the notion of layers and interpreted code, as in [9], to combine texture instances in arbitrary ways. This capability would allow the generation of arbitrarily complex infinite resolution textures from simple primitives and operators.

We also plan to build up a family of texture generators for simulating terrains of architectural interest. This would include treetops, shrubbery, the appearance of rows of houses – including slanted roofs and chimneys and even roof texture – and so on. The idea is that an Architect could work up a sketch of a landscaping, or cluster of dwellings, or a city, using broad strokes – perhaps just to play with the feel of how various arrangements would look.

From a production standpoint, it would be good to have a complete environment for designing multiresolution procedural textures. This would include programming, testing and debugging facilities that can work together with the paint system.

A final area of investigation is the use of techniques for analyzing the correlation between levels of the bandpass pyramid described in Section 3 in order to perform extrapolation of coarse resolution images when detail information is not available.

Acknowledgements

The authors wish to thank Lance Williams for fruitful discussions and for pointing out the work of [8]. Many thanks also to Stephane Mallat for several valuable suggestions. Special thanks to Karl Sims for inspiring comments on multiscale noise.

This work was partially supported by grants from MCT/CNPq – Conselho Nacional de Desenvolvimento Científico e Tecnológico and from the National Science Foundation.

REFERENCES

[1] Deborah Berman, Jason Bartell, and David Salesin. Multiresolution painting and compositing. *Computer Graphics, Annual Conference Series (SIGGRAPH '94 Proceedings)*, pages 85–90, 1994.

[2] Peter J. Burt. The laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, 31:532–540, April 1983.

[3] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, June 1982.

[4] John Peter Lewis. Texture synthesis for digital painting. *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18(3):245–252, July 1984.

[5] Lively pictures. *Byte Magazine*, 20(1):171–174, 1995. Live Picture – Product Review.

[6] Stephane Mallat. Multifrequency channel decompositions of images and wavelet models. *IEEE Trans. on Acoust. Signal Speech Process.*, 37(12):2091–2110, 1989.

[7] Joan M. Ogden, Edward H. Adelson, J.R. Bergen, and Peter J. Burt. Pyramid-based computer graphics. *RCA Engineer*, 30(5):4–13, September–October 1985.

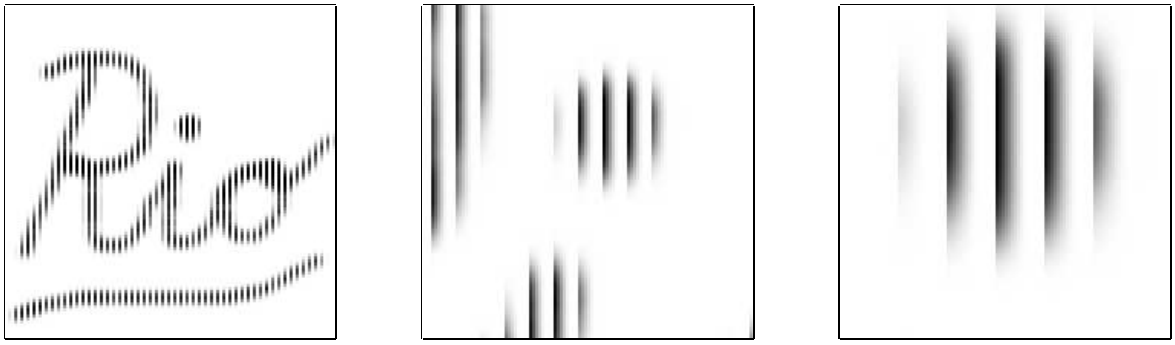
[8] Alex Pentland and Bradley Horowitz. A practical approach to fractal-based image compression. In *Proceedings of Data Compression Conference*, pages 176–185, held in Snowbird, UT, 1991. IEEE Computer Society Press.

[9] Ken Perlin. An image synthesizer. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):287–293, 1985.

[10] Ken Perlin and Luiz Velho. A wavelet representation for unbounded resolution painting. Technical report, New York University, New York, 1992.

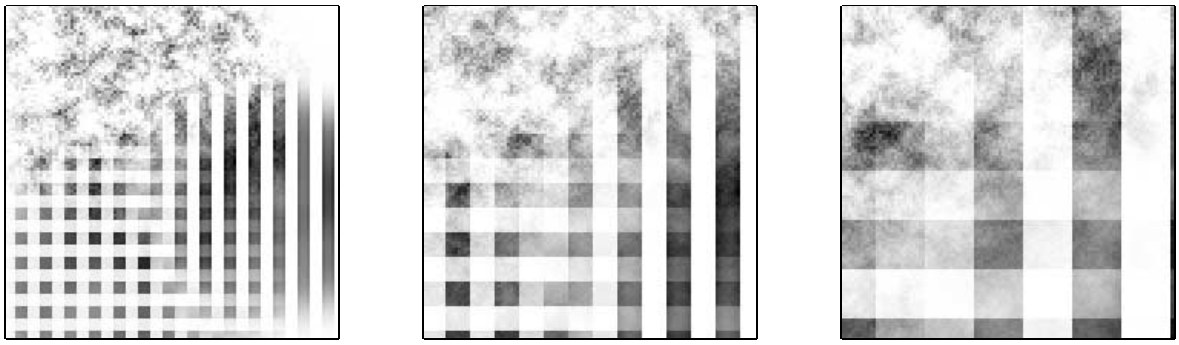
[11] Lance Williams. Pyramidal parametrics. *Computer Graphics (SIGGRAPH '83 Proceedings)*, 17(3):1–11, July 1983.

[12] Xres, the alternative to photoshop? *Mac Format Magazine*, 23, pages 72–74, 1995. XRes – Graphics Software Review.



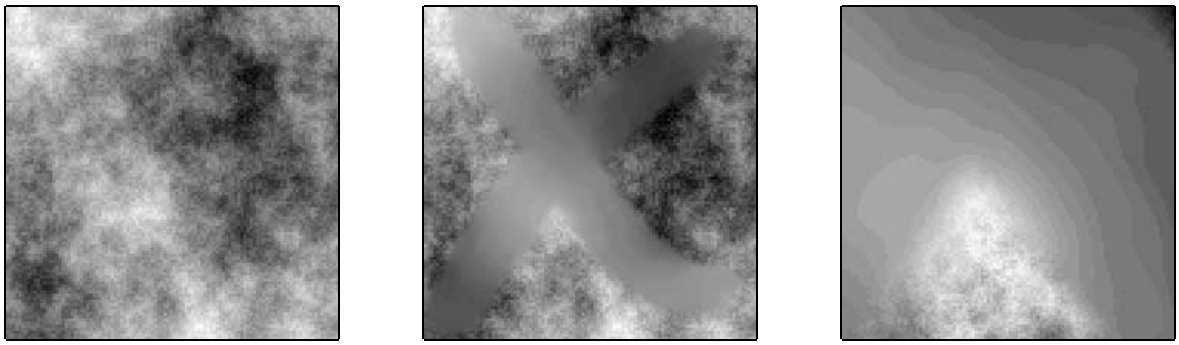
(a) (b) (c)

Figure 2: Sawtooth Texture



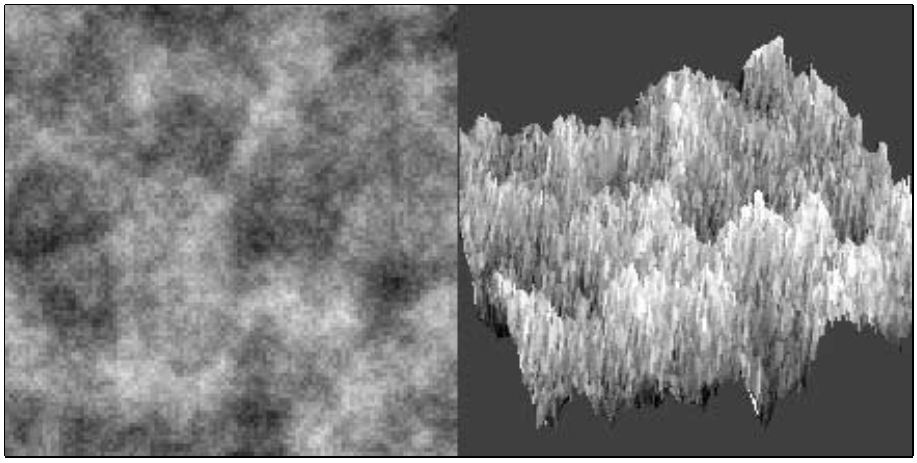
(a) (b) (c)

Figure 3: Blending of rock, squares, and horizontal stripes

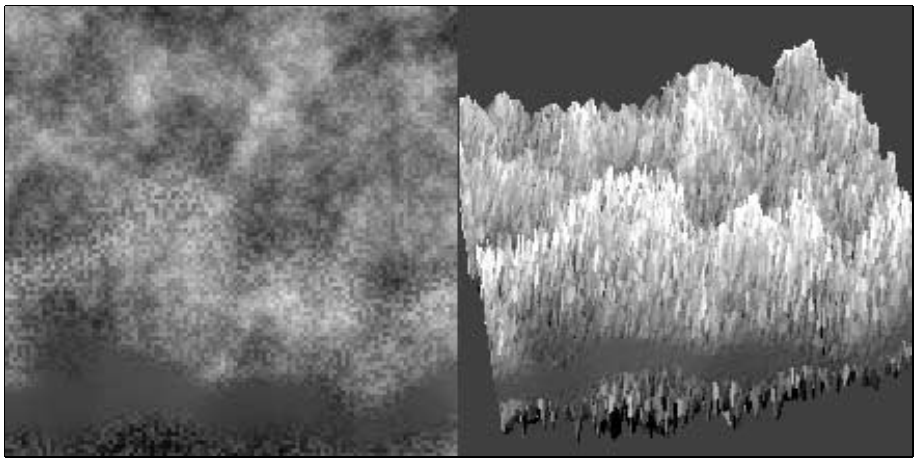


(a) (b) (c)

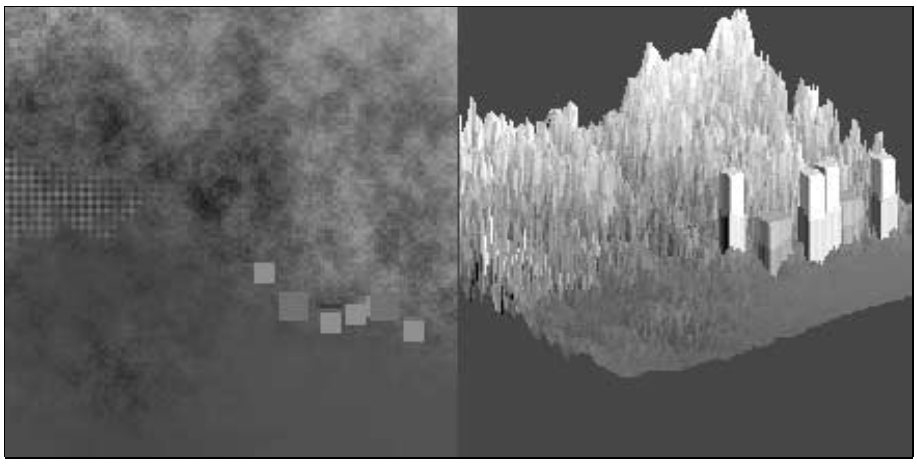
Figure 4: The effect of a smoothing brush



(a)



(b)



(c)

Figure 5: Interactive design of a terrain model