Automatic Synthesis of Isotropic Textures on the Sphere from Sample Images Using the Laplacian Pyramid Transform

Curtis Sirl-Moore
Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131

Lance R. Williams
Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131

June 8, 2001

Abstract

A method for synthesizing a given texture sample directly on the surface of a three-dimensional object is presented. The method combines the Heeger-Bergen texture synthesis algorithm with an implementation of the Laplacian pyramid transform on the sphere. It avoids the problems of distortion and discontinuity inherent in texture mapping by synthesizing texture directly on the object surface. Furthermore, because it is automatic, no artistic skill is required and no *ad hoc* parameter-tweaking is necessary. Finally, while it takes several iterations to terminate, it is notably inexpensive to compute.

1 Introduction

1.1 Motivation

Computer-generated scenes have become increasingly prevalent in the film, computer gaming, and virtual reality industries. As computer graphics hardware and software become more sophisticated, the need to add realism to these scenes becomes more compelling. One way to make computer-rendered objects more realistic is to add texture to their surfaces. To date, all the techniques for doing this on three-dimensional surfaces have been inadequate in some fashion.

1.2 Techniques for Adding Texture to Surfaces

There are several techniques typically used to add texture to surfaces in computer-generated scenes. *Texture mapping* is the process of projecting a flat texture image onto an arbitrary three-dimensional surface [6]. Texture mapping can be visualized as taking a rubber sheet with a texture printed on it, and stretching and warping the sheet onto a wireframe model of the surface. The problems inherent in this approach are *distortion* and *discontinuity*. The former occurs when the sheet must be non-uniformly stretched to fit the surface (e.g., in the case of a surface with non-zero curvature). The latter occurs because there exists no *diffeomorphism*, or smooth one-to-one mapping, between the plane and the surface of a smooth solid object. Because of these problems with texture mapping, several techniques have been employed to place texture *directly* on surfaces.

One alternative to texture mapping is to manually "paint" the texture onto the three-dimensional object using a computer program [5]. This of course requires artistic skill and is time consuming. Another technique is *procedural texture generation*, which adds texture to a surface based on simple synthesis models (e.g., reaction-diffusion and fractals) [3, 11]. While the results are often very interesting, they cannot be controlled *a priori* to produce a specific texture, so one must experiment with the parameters to the procedure until a satisfactory texture is generated. Finally, *solid textures* [8] can be generated using a three-dimensional array of color values and assigning the colors of points on the surface based on their location in the array. In this case the mapping problem is completely avoided because there is no projection.

However, solid texturing tends to be computationally expensive.

1.3 Texture Synthesis from Samples

Recently, researchers have developed algorithms which attempt to synthesize a given *texture sample* by generating an image with the same textural properties. Such *sample-based synthesis* is attractive because it is completely automatic, requiring no artistic skill and no *ad hoc* parameter adjustment to achieve a desired effect. Furthermore, this type of synthesis is often implemented using fast transforms (e.g. the Laplacian and steerable pyramids), so computational costs are minimal.

Heeger and Bergen [7] described an algorithm which first analyzes a sample texture using a set of filters, then operates on a white noise image until it produces the same distribution of responses to the filters. The guiding hypothesis is that the filters are analogous to receptive fields of neurons in the human visual system, so two images will look the same to a human subject if they elicit similar responses. The goal is to create an image which produces the same distribution of responses, and this is achieved through an iterative process. The Heeger-Bergen algorithm consists of two high level stages: *analysis*, in which filter responses are computed using an *image pyramid* decomposition of the texture sample, and *synthesis*, in which both the gray levels and the pyramid transform coefficients of a random noise image are transformed until they have the same distributions as those of the input texture sample and its pyramid representation.

1.4 Summary of Solution

This paper describes an algorithm for synthesizing a given sample texture directly on the 2D curved surface which forms the boundary of a 3D object. The procedure combines the Heeger-Bergen texture synthesis algorithm with an implementation of the Laplacian pyramid transform on the sphere. The *analysis* and *synthesis* stages are like those of the Heeger-Bergen algorithm, except that instead of coercing a random noise image to have the same statistics as the texture sample, a random noise *sphere* is coerced.

2 The Heeger-Bergen Algorithm

2.1 Introduction

The texture synthesis algorithm proposed by Heeger and Bergen [7] works well on a wide range of sample textures. The key components of the procedure are the *steerable pyramid transform*, *histogram matching*, and the *KL transform*.

2.2 The Steerable Pyramid

Heeger-Bergen uses the filters in the *steerable pyramid* transform [4] to represent the response of the human visual system to the sample texture. The pyramid transform decomposes an input image into a series of *subbands*, each representing the response of a filter tuned to image elements that have a specific size and orientation. The algorithm operates on a random noise image, iteratively transforming its gray levels and pyramid subbands until they have the same distributions as those of the input texture sample.

2.3 Histogram Matching

In order to force the pyramid decomposition of the output to have the same distribution as that of the input, the histogram of each of the noise image's subbands is matched to that of the sample image's subbands. However, doing this alters the gray-level histogram of the reconstructed image, so once the pyramid transform is inverted, the histogram of the result is matched to that of the sample texture. To avoid this problem, this set of steps (expand, match, collapse, match) is repeated several times until the output image looks like the given texture sample. An initial gray-level histogram match takes place before this iterative process begins. The process converges in practice, though this is not guaranteed in general.

2.4 The KL Transform

In order for the algorithm to synthesize a color texture, it must perform the iterative grayscale matching procedure described above on the three color bands (red, green, and blue) of the sample texture. However, these three bands are typically statistically correlated, so operating on each independently will introduce color artifacts in the resulting synthetic texture. To get around this problem, each band is treated as a 3D Gaussian random variable. The bands can then be *decorrelated* using the KL transform [10], which computes a basis in which the correlation matrix is diagonal. The basis vectors are the *eigenvectors* of the sample's covariance matrix. These vectors form the rows of an orthogonal decorrelating matrix, M, by which the $3 \times N$ sample matrix is multiplied. The resulting three bands have zero covariance (i.e. are "uncorrelated" to second order), and can thus be operated on separately. The iterative process is then applied (beginning with three random noise images). Once finished, the KL transform is inverted by multiplying the $3 \times N$ output matrix by M^{-1}. This process results in three synthesized texture images, which are the red, green, and blue components of the output.

3 The Laplacian Pyramid

3.1 Introduction

The Laplacian pyramid [1] is a linear transform that represents an image by decomposing it into a set of bandpass images and a single lowpass image. It is a predecessor to the fast wavelet transform.

3.2 Analysis

The Laplacian pyramid transform decomposes an input image, I, into a set of bandpass images, B_i, and a single lowpass image, L_k. The fundamental operations in the pyramid decomposition process are *reduce* and *project*. The *reduce* operation corresponds to convolution with a Gaussian kernel followed by downsampling by a factor of two, while *project* upsamples by a factor of two and then interpolates by convolution with a Gaussian kernel. Given these two operations, the Laplacian pyramid decomposition of I can be computed as follows: Let $L_1 = \text{reduce}(I)$. The *reduce* operation removes high-frequency content, so L_1 is lowpass. Now, let $B_1 = I - \text{project}(L_1)$. Figure 1 is a block diagram of this computation. Note that I can be perfectly reconstructed solely from B_1 and L_1:

$$I = B_1 + \text{project}(L_1).$$

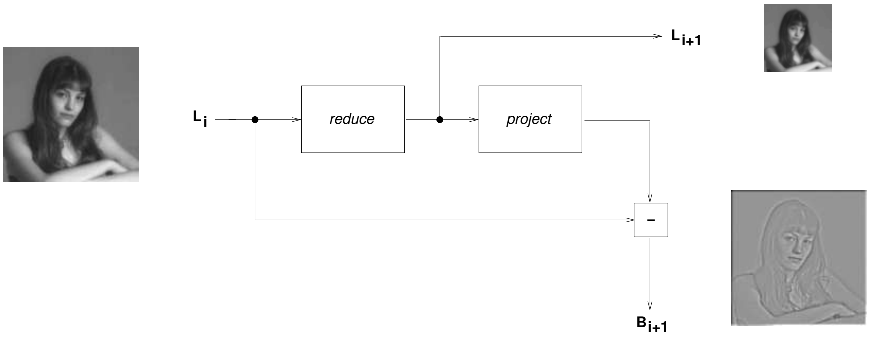

Figure 1: Laplacian pyramid analysis.

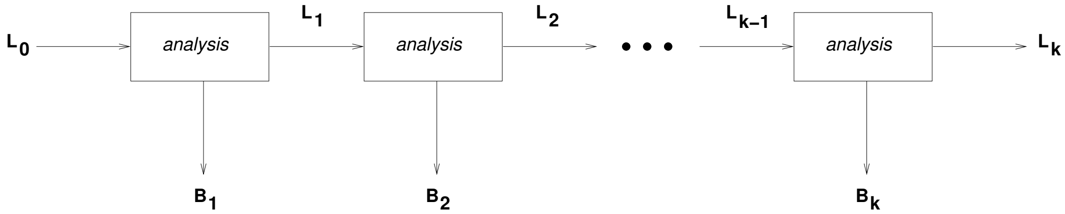

Figure 2: Expanding a k-level Laplacian pyramid.

A pyramid of k levels can be computed recursively:

$$
\begin{aligned}
L_i &= \text{reduce}(L_{i-1}) \\
B_i &= L_{i-1} - \text{project}(L_i).
\end{aligned}
$$

The process of computing the bandpass images, B_i, and the final lowpass image, L_k, is often implemented as a single procedure, *expand*, which takes an image, I, and a number of levels, k, as input. The *expand* procedure outputs the k-level Laplacian pyramid decomposition of I (Figure 2). Figure 3 shows a sample pyramid decomposition.

3.3 Synthesis

Because each B_i is a difference image, it contains all of the information present in L_{i-1} that is no longer present in L_i. Therefore the intermediate lowpass images, L_i, are redundant, and I can be reconstructed using only the final lowpass image L_k and the set of bandpass images B_i. The process of reconstructing I from the bandpass images B_i and the final lowpass image

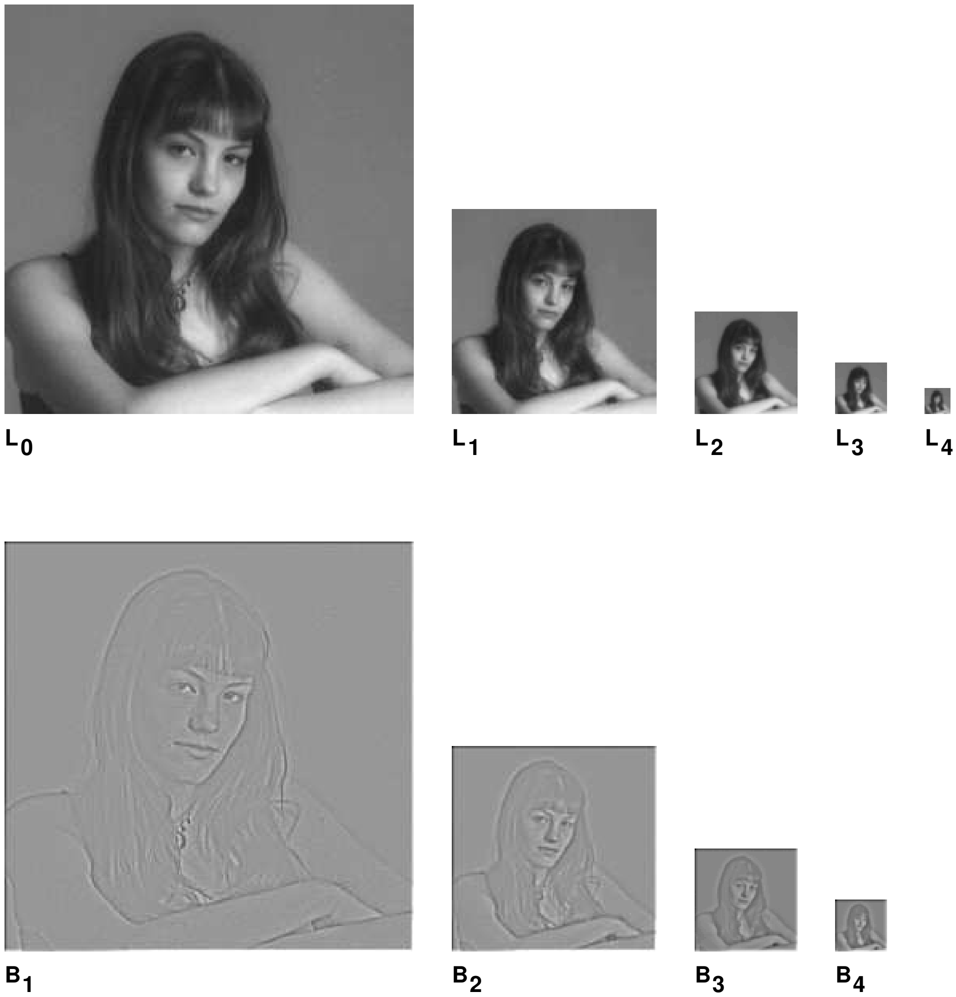

Figure 3: Example Laplacian pyramid decomposition ($k = 4$).

L_k is the *synthesis* side of the Laplacian pyramid transform. The algorithm is as follows: Let $L'_{k-1} = B_k + \text{project}(L_k)$. Proceed in this manner until L'_0 is computed. Figure 4 is a block diagram of this computation. Assuming the B_i's and L_k have not been altered, L'_0 will be the original input image, I. Synthesis is often implemented as a single procedure, *collapse*, which takes the B_i's and L_k as input, and repeatedly projects and adds its intermediate results, as previously described, until L'_0 ($\approx L_0$) has been recovered (Figure 5).

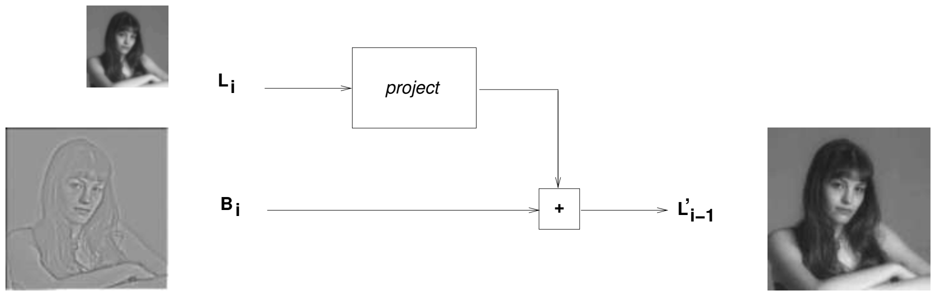

Figure 4: Laplacian pyramid synthesis.

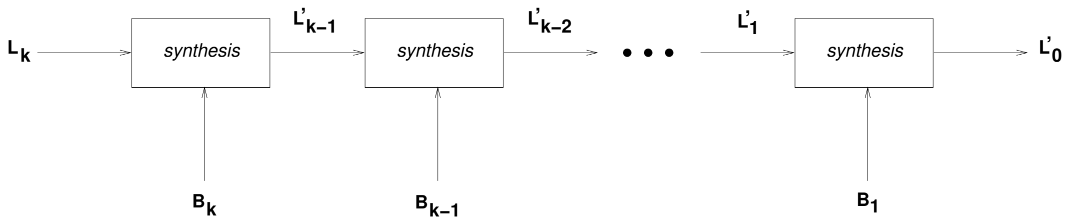

Figure 5: Collapsing a k-level Laplacian pyramid.

4 The Laplacian Pyramid on the Sphere

4.1 Introduction

In order to perform texture synthesis on arbitrary surfaces using an approach like that of Heeger and Bergen, the definition of an image pyramid must be extended to the surfaces of three-dimensional objects. In the case of the

Laplacian pyramid, the basic operations of downsampling, upsampling, and convolution must be redefined in order to arrive at new definitions for *reduce* and *project*. This section explains how these operations can be defined in order to implement a Laplacian pyramid on a sphere.

4.2 Recursive Subdivision

In computer graphics, the surfaces of three-dimensional objects are rendered as sets of three-sided planar faces, and an object is defined by specifying the vertices and reflectances of these triangles. In order to achieve realism, an object often must be rendered using millions of triangles. In the case of a highly curved surface, like that of a sphere, the technique of *recursive subdivision* can be used to increase the number of triangles that compose the object and thereby increase the realism of its rendered surface.

The idea is rather simple: Start with a three-dimensional object and repeatedly divide each of its triangles into four smaller triangles, projecting the new vertices to the object surface, until a desired measure of accuracy (typically surface curvature or depth of recursion) is reached. There are several methods for subdividing a triangle, including *bisection*, *center of mass*, or *edge midpoints*. The implementation discussed in this paper uses the last method. Let $\vec{v}_0$, $\vec{v}_1$, and $\vec{v}_2$ be vectors from the center of the sphere to the vertices of the triangle. Compute vectors to the midpoints of each of the triangle's edges:

$$\vec{m}_{01} = \frac{\vec{v}_0 + \vec{v}_1}{2}, \qquad \vec{m}_{02} = \frac{\vec{v}_0 + \vec{v}_2}{2}, \qquad \vec{m}_{12} = \frac{\vec{v}_1 + \vec{v}_2}{2}.$$

Now, extend these three new vectors to the surface of the sphere. In the case of a unit sphere, this can be implemented by normalizing each vector to have unit length:

$$\vec{m}'_{01} = \frac{\vec{m}_{01}}{\| \vec{m}_{01} \|}, \qquad \vec{m}'_{02} = \frac{\vec{m}_{02}}{\| \vec{m}_{02} \|}, \qquad \vec{m}'_{12} = \frac{\vec{m}_{12}}{\| \vec{m}_{12} \|}.$$

This process yields three new vertices, allowing the parent triangle to be divided into four child triangles (see Figure 6). Note that, in order for recursive subdivision to work as defined above, the initial sphere approximation must consist only of triangles.

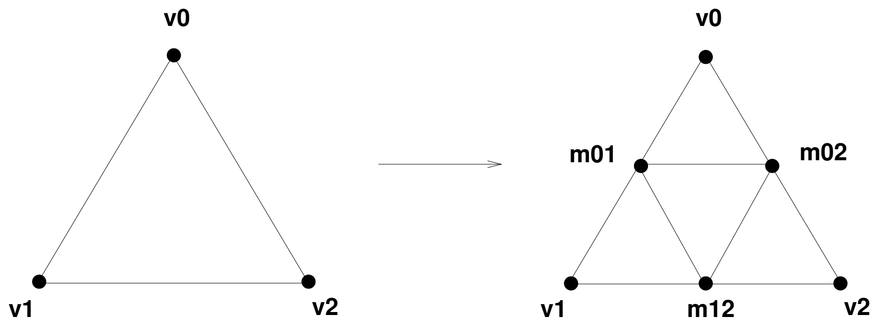

Figure 6: Subdividing a triangle using edge midpoints.

4.3 Resampling

Given a technique for recursive subdivision as described above, it is now possible to define downsampling and upsampling operations on the sphere.

Downsampling a sphere is rather straightforward. Given a sphere of $N \times 4^k$ faces, the output is a sphere of $N \times 4^{k-1}$ faces. The value of each face i in the downsampled sphere is simply the value of face $4i$ in the original sphere:

$$v_{k-1}(i) = v_k(4i).$$

In other words, the value of every fourth face is kept and the others are discarded.

The upsampling operation is more involved. It takes a *parent* sphere of $N \times 4^k$ faces as input, and produces a *child* sphere of $N \times 4^{k+1}$ faces (where N is the number of faces in the initial sphere approximation). Each face i in the parent sphere becomes 4 faces ($4i,\ 4i+1,\ 4i+2,\ 4i+3$) in the child sphere, using subdivision as described above. Now a value must be defined for each child face based on the value of the parent face. In order to produce a desirable distribution of values,[1] we define a binary-valued *upsampling function*, s, on faces. s_0 labels the faces of the initial sphere approximation so that their distribution is preserved during upsampling. The constraint that

$$s_0(j) = 1 - s_0(i) \qquad \text{for all } j \text{ adjacent to } i$$

[1] The goal is to produce a distribution of values in which 50% of the faces in the child sphere have the value of their parents, while the other 50% are zero.

ensures an even distribution that will not be compromised by subdivision. Note that in order for this condition to be met, all the vertices of the initial sphere approximation must have even degree.[2]

In general, when a sphere of $N \times 4^k$ faces is upsampled to produce a sphere of $N \times 4^{k+1}$ faces, the labels for the child sphere are defined in terms of the parent:

$$s_{k+1}(4i+j) = \begin{cases} 1 - s_k(i) & \text{for } j = 0 \\ s_k(i) & \text{for } j \in \{1,2,3\} \end{cases}$$

where $s_k(i)$ is the label of face i in a sphere with $N \times 4^k$ faces (the parent). The values of $s_{k+1}(4i+j)$ are the labels of face i's children in the upsampled sphere (Figure 7). It can be shown by induction that the adjacency constraint is still satisfied:

$$s_k(j) = 1 - s_k(i) \qquad \text{for all } j \text{ adjacent to } i.$$

Now the value of each child face can be computed:

$$v_{k+1}(4i+j) = s_{k+1}(4i+j)\ v_k(i) \qquad \text{for } j \in \{0,1,2,3\}$$

where $v_k(i)$ is the value of face i in the parent sphere.

4.4 Gaussian Convolution

In the case of a two-dimensional image, the value of a pixel after discrete convolution with a given kernel can be computed as a weighted sum of the values of the pixel's neighbors, where the kernel defines the weighting function. The computation is straightforward because all of the pixels are the same size and their spacing is uniform. In the case of a sphere, convolution must be redefined to take into account the variable size and spacing of neighboring triangular faces. In our implementation, the area, a_j, of face, j, is used as a measure of its size, and the difference in angle, $\cos^{-1}(\vec{u}_i \cdot \vec{u}_j)$, between the faces' centers, $\vec{u}_i$ and $\vec{u}_j$, defines the faces' distance.

[2]For this reason, the implementation discussed in this paper uses an octahedron (8 sides) as an initial approximation. Although the icosahedron (20 sides) produces a more regular subdivision, its vertices do not have even degree, so s_0 cannot be defined on its faces.

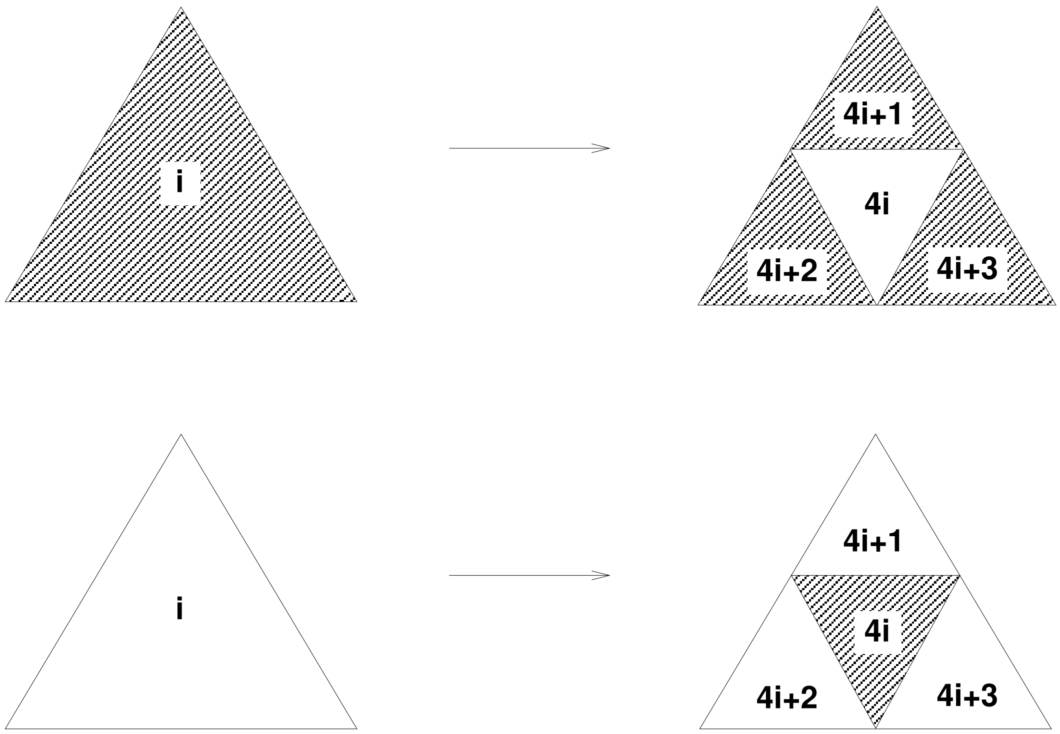

Figure 7: The upsampling function. $s = 0$ for shaded faces and 1 for unshaded faces.

In order to compute the value of a face after convolution with a Gaussian kernel, we must define the kernel weights of faces in its neighborhood. The d-neighborhood, $N_d(i)$, of face i, is the set consisting of face i, all the faces adjacent to face i, all the faces adjacent to these faces, etc., to a depth of d. In our implementation, we use $d = 4$ to define the neighborhood. The kernel weight of face j relative to face i for $j \in N_4(i)$ is:

$$W_{ij} = G\left(\frac{\cos^{-1}(\vec{u}_i \cdot \vec{u}_j)}{\sigma}\right) a_j$$

where $G(x) = e^{-x^2/2}$ and

$$\sigma = \overset{\max}{<k, \ell \in N_1(k)>} \cos^{-1}(\vec{u}_k \cdot \vec{u}_\ell)$$

In the expression for W_{ij}, dividing $\cos^{-1}(\vec{u}_i \cdot \vec{u}_j)$ by σ dilates G, a Gaussian of unit standard deviation, so that it is (in effect) a Gaussian of standard deviation, σ. The standard deviation of the Gaussian at each level of the pyramid therefore equals the maximum distance between the centers of any two adjacent faces of the sphere at that level. This is a conservative design choice (compared to the alternative of setting σ equal to the average distance between the centers of faces) which minimizes the aliasing introduced by downsampling, yet remains faithful to Burt's formulation[1]. As in the case of a two-dimensional image, we want the sum of the kernel weights to be one, so we divide the convolution by the factor, $\sum_{j \in N_4(i)} W_{ij}$. The value of face i after convolution is therefore:

$$v'_i = \frac{\sum_{j \in N_4(i)} W_{ij} v_j}{\sum_{j \in N_4(i)} W_{ij}}$$

where v_j is the value of face j before convolution.

4.5 Reduce and Project

Now that resampling and convolution have been defined for the sphere, suitable definitions for *reduce* and *project* can be given. While *reduce* is straightforward (first convolve the input sphere with a Gaussian and then downsample the result), *project* must apply a normalization factor to ensure the correct range of values in its result. Recall that when a sphere is upsampled,

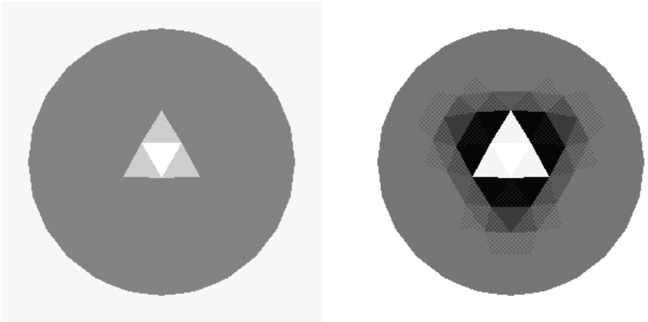

Figure 8: Gaussian and Laplacian kernel impulse responses.

only half of the faces of the child sphere inherit the value of their parent; the rest are zero valued. The convolution of such an upsampled function with a Gaussian will have a lower average value than the input. In order to preserve the average value of the input, we divide the convolution by the factor, $\sum_{j \in N_4(i)} W_{ij} s_j$. The value of face i after upsampling and convolution is therefore:

$$v_i' = \frac{\sum_{j \in N_4(i)} W_{ij} s_j v_j}{\sum_{j \in N_4(i)} W_{ij} s_j}.$$

The normalization factor is derived by considering the effect of upsampling and Gaussian convolution on a function of constant value, C. Since for a function of constant value, v_j equals C, dividing by $\sum_{j \in N_4(i)} W_{ij} s_j$ ensures that after upsampling and Gaussian convolution, v_i' also equals C.

4.6 Examples

Figure 8 shows the impulse responses of the kernels used to construct the lowpass and bandpass components of the Laplacian pyramid on the sphere. The Gaussian (lowpass) kernel is shown on the left and the effective Laplacian (bandpass) kernel on the right. For purposes of display, the values of each sphere were first normalized to $[0, 1]$, and then a linear grayscale transformation, $y = 3.64x$, was applied. Out-of-range values have been clipped. Figure 9 shows an example Laplacian pyramid decomposition on the sphere. Each sphere has been resized to have a radius of 1 and values have been normalized to $[0, 1]$ for display.

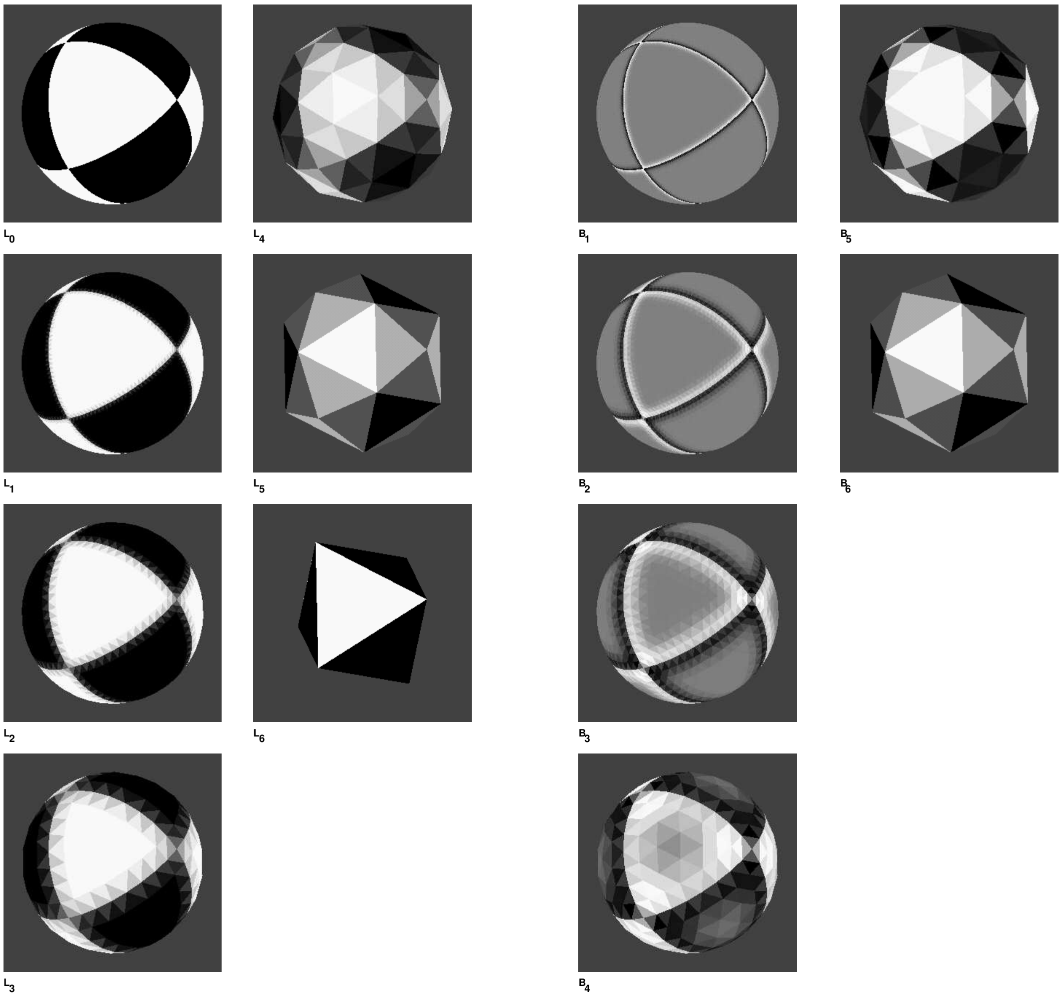

Figure 9: Example Laplacian pyramid decomposition on the sphere ($k = 6$).

4.7 Experimental Results

This section contains synthesis results obtained using the Heeger-Bergen texture synthesis algorithm combined with our implementation of the Laplacian pyramid on the sphere. Figures 10 and 11 depict synthesis pairs. The input to our implementation is shown on the left side of each pair, the resulting synthetic texture on the right. In both figures, the input textures are 128×128-pixel images of various foods, obtained using a digital camera attached to the eyepiece of a dissecting microscope. The Laplacian pyramid decomposition of each sample texture was computed with $k = 5$. The initial sphere approximation consisted of 128 faces, and was obtained by recursively subdividing an octahedron twice. Each output synthesized texture sphere therefore consists of $128 \times 4^5 = 131,072$ faces. The results shown were obtained after 2 iterations of the Heeger-Bergen method.

The synthesized spheres in Figure 10 successfully capture the essential textural characteristics of their inputs, due to the fact that the textural elements in these samples are *isotropic* (rotation-invariant) and *homogeneous* (uniformly distributed). The convolution kernels in the Laplacian pyramid are rotationally symmetric, and successfully capture the textural qualities of the input. The results shown in Figure 11 are not as satisfactory, however, because these texture samples are anisotropic and presumably contain higher-order statistics than those used by the Heeger-Bergen algorithm (e.g., correlation between information at different scales).

Figure 12 depicts synthesis results using the moons of Jupiter (top row) as input. The inputs to our implementation (middle row) were images of the surfaces of, from left to right, Io (128×128), Europa (64×64), Ganymede (128×128), and Callisto (128×128). These images were obtained from the NASA Galileo website (http://www.jpl.nasa.gov/galileo/index.html). Synthesis results are shown at the bottom of the figure.

4.8 Extensions and Future Work

The discussion here has been restricted to synthesis on the sphere, mainly to emphasize the fact that the problem of *discontinuity*, which occurs during texture mapping onto a closed surface, has been circumvented. The algorithm presented in this paper can be extended to work on any arbitrary

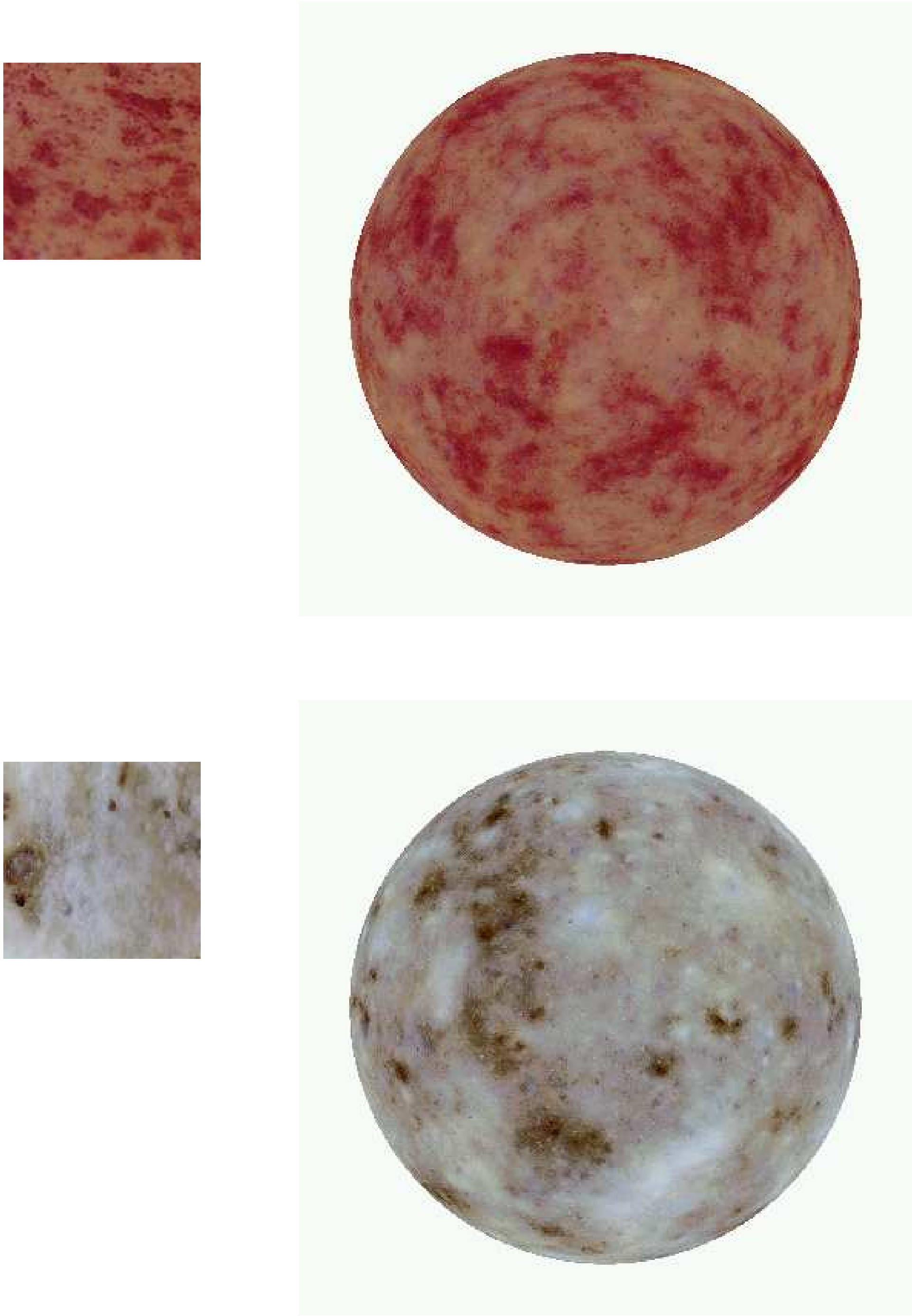

Figure 10: Successful synthesis of apple skin (top) and tortilla chip (bottom) textures.

Figure 11: Failure examples: avocado and pear skin.

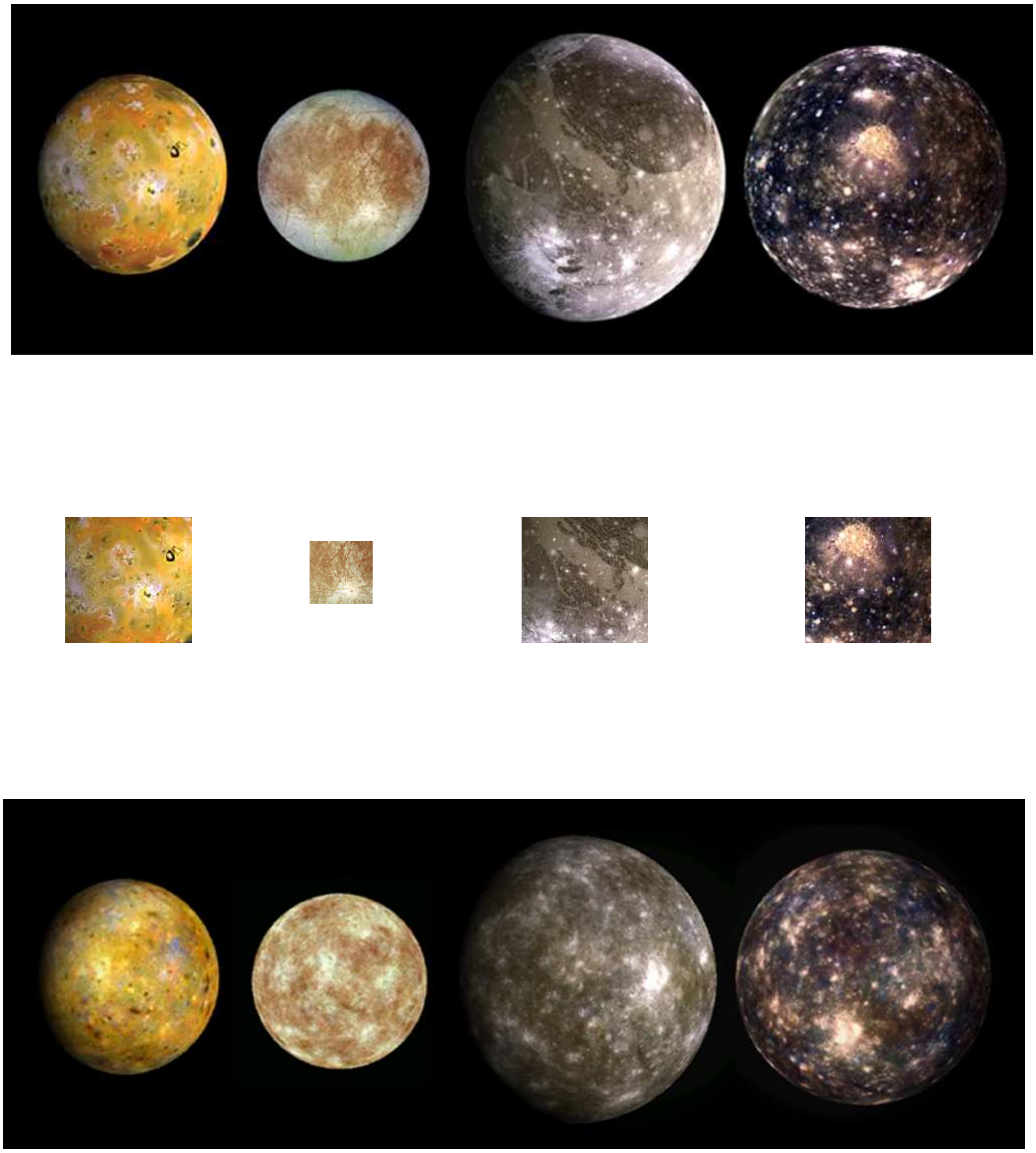

Figure 12: Synthesized moons of Jupiter.

three-dimensional surface, given a procedure for performing resampling and convolution on that surface. Note that for the recursive subdivision algorithm described to work the surface tessellation must consist only of triangles. All the vertices in this initial triangulation must also have even degree in order for resampling to work as defined.

As noted previously, the Heeger-Bergen algorithm works by decomposing the input texture sample using the *steerable pyramid*, not the Laplacian pyramid. Presumably an even wider range of textures could be synthesized on three-dimensional surfaces once a definition is provided for the steerable pyramid on the sphere.

Finally, the Heeger-Bergen method only matches the *first-order* statistics of the pyramid filter responses. An implementation which takes into account higher-order statistics [2, 9] will be able to synthesize an even wider range of input textures.

4.9 Conclusion

This paper describes a method for synthesizing a given texture sample on the surface of a three-dimensional object. The method combines an implementation of the Laplacian pyramid on the sphere with the Heeger-Bergen method for texture synthesis from samples. Because synthesis occurs directly on the object surface, the problems inherent in texture mapping (distortion and discontinuity) are avoided. Experimental results of synthesis on the surface of a sphere indicate that the algorithm runs quickly and works well on a broad range of textures.

References

[1] Burt, P., Fast Filter Transforms for Image Processing. *Computer Graphics and Image Processing 16* (1981), pp. 20-51.

[2] De Bonet, J.S., Multiresolution Sampling Procedure for Analysis and Synthesis of Texture Images, Proceedings of ACM SIGGRAPH 97 (1997).

[3] Fournier, A., Fussel, D., and L. Carpenter, Computer Renderings of Stochastic Models. *Communications of the ACM 25* (1982), pp. 371-384.

[4] Freeman, W.T. and E.H. Adelson, The Design and Use of Steerable Filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence 13* (1991), pp. 891-906.

[5] Hanrahan, P. and P. Haeberli, Direct WYSIWYG Painting and Texturing of 3D Shapes. Proceedings of SIGGRAPH 90. In *Computer Graphics* (1990), Vol. 24, ACM SIGGRAPH, pp. 215-223.

[6] Heckbert, P.S., Survey of Texture Mapping. *IEEE Computer Graphics and Applications 6* (1986), pp. 56-67.

[7] Heeger, D.J. and J.R. Bergen, Pyramid-Based Texture Analysis/Synthesis, Proceedings of SIGGRAPH 95 (1995), pp. 229-238.

[8] Peachy, D.R., Solid Texturing of Complex Surfaces. Proceedings of SIGGRAPH 85. In *Computer Graphics* (1985), Vol. 19, ACM SIGGRAPH, pp. 279-286.

[9] Portilla, J. and E.P. Simoncelli, Texture Modeling and Synthesis using Joint Statistics of Complex Wavelet Coefficients, *IEEE Workshop on Statistical and Computational Theories of Vision*, Fort Collins, CO, 1999.

[10] Pratt, W.K., *Digital Image Processing*, John Wiley & Sons, New York, 1991.

[11] Turk, G., Generating Textures on Arbitrary Surfaces Using Reaction-Diffusion. Proceedings of SIGGRAPH 91. In *Computer Graphics* (1991), Vol. 25, ACM SIGGRAPH 91, pp. 289-298.