Multiresolution Textures from Image Sequences

Eyal Ofek, Erez Shilat, Ari Rappoport, and Michael Werman
The Hebrew University of Jerusalem

Extracting textures from image sequences eliminates perspective distortions and removes highlights and reflections to produce high-quality multiresolution images with accurate color.

Rendering is one of the most important tasks in computer graphics and animation, and texture maps add essential visual content to the rendered image. Unfortunately, extracting textures from a single photograph poses severe difficulties and is sometimes impossible, while artificial texture synthesis does not address the full range of desired textures.

We present a method for computing high-quality, multiresolution textures from an image sequence. Our technique has the following features:

1. It can be used with images in which the textures are present in different resolutions and different perspective distortions.
2. It can extract textures from objects with any known 3D geometric structure; specifically, we are not restricted to planar textures.
3. It removes directional illumination artifacts such as highlights and reflections.
4. It stores the resulting texture efficiently in a multiresolution data structure.
5. It imposes no restrictions on the computed texture, which can be a constant color or richly colored.

One particularly attractive application of our method, illustrated in the conclusion, produces animation sequences of existing objects endowed with synthetic behavior. This emphasizes the advantage of the multiresolution representation, since each frame uses the level of detail it needs for any location in the texture.

Creating textures for realistic images

In addition to the geometry of the rendered object, high-quality rendering requires the object's material properties, texture maps associated with the object's surfaces, and algorithms to produce images given these data. The simulation of light is well understood.[1] Artificial texture generation, on the other hand, still involves some unsolved difficulties. Although specific textures have been simulated successfully, artificially generated textures usually look too "clean," even when noise is added using statistical methods. In practice, you must use textures from real images (such as photographs or video stills) or textures created using 2D paint systems to create satisfactory visual results.

Certain applications inherently require textures from real images. For example, to endow a real object with a personality and orchestrate its motion with animation software, we would naturally want its real textures to be available to the rendering system. But computing texture maps from a real image poses several difficulties:

- The lighting conditions under which the image was taken might not match the desired illumination on the texture-mapped object. Specifically, the image usually contains specular light effects such as highlights and reflections that are not supposed to be seen when the texture is mapped onto an object.
- The texture's geometry as captured in the image might not match that needed by the renderer's texture mapping algorithm. Textures are usually captured distorted due to perspective, while they are needed in the unit square.
- The image quality might not be sufficient, lacking spatial or color resolution or containing a high level of noise. This may be especially true for video stills.

To counter these difficulties, we might try to control the photographic conditions—often a difficult task. Moreover, for many common types of textures, controlled photography is impossible. Examples include inconveniently located textures (such as outdoors), illumination artifacts inherent in the object's material (such as an engraving on a mirror), or textures requiring multiple images to capture (such as a long wall in a narrow corridor or a building hidden by trees).

0272-1716/97/$10.00 © 1997 IEEE

Using multiple images is attractive because it lets us use existing film footage that was not taken expressly to produce textures. However, this presents us with a new set of problems:

- The geometry of the texture might differ in each image.
- The resolution of the texture might differ in each image.
- Light components based on viewing direction, such as highlights and reflections, might differ in each image.

We present here a method to compute high-quality multiresolution 2D textures from multiple images that overcomes these problems.

Related work

Burt and Kolczynski[2] presented an algorithm to fuse several images into a single image. The algorithm is limited to images taken by a stationary camera and mostly suits fusion of images obtained from sensors of different natures. Irani and Peleg[3] showed how to create a super-resolution image from an image sequence when the relative translations between the images are known to subpixel accuracy. Their algorithm assumes only 2D translations, requires a priori determination of the final resolution, and is computationally intensive. Combining different resolutions in the same setting is usually done through a multiresolution representation, such as a pyramid or a quadtree.

Berman et al[4] described a system that uses a wavelet quadtree to economically store multiresolution information. The application is essentially a paint system in which new paint covers (or is blended with) former information. This does not suit combining several images that depict the same object, since the final image obtained depends on the order of fusion of the images and a wavelet pyramid cannot represent the intermediate data structures essential for such fusion.

For highlight recognition and removal from a single image, polarization[5] and color space techniques have been proposed.[6] Polarization gives promising results but requires photography using a special filter. Color space methods use the ideas behind Shafer's dielectric material model.[7] Pixels corresponding to a dielectric material's color are concentrated near a line segment in color space, while pixels corresponding to the highlights deviate from this segment. This serves to isolate the material color's segment. These methods do not work well when the surface is not smooth or is highly textured, or when there is more than a single light source.

Lee and Bajcsy detected nonoverlapping highlight regions in pairs of images by matching pixels with similar colors.[8] Their method cannot distinguish between highlight and occlusion and does not accurately specify the highlight regions. None of the color space methods works for richly colored textures.

The proposed method

We can compute high-quality multiresolution textures from an image sequence. The input consists of a set of images and a mask on each image denoting the portion of the image from which texture should be extracted.

Obtaining such a mask completely automatically is, of course, a very difficult problem whose solution is a primary goal of the large field of computer vision. Generally, we want to save the user as much work as possible while allowing for human intervention to correct the system mistakes.

One possible scenario lets the user mark a set of points whose 3D structure is known; the system then tracks these points along the image sequence to produce the necessary masks. Alternatively, the system could automatically compute the points' 3D structure according to well-known epipolar geometry equations. Asking users to mark an initial object makes sense because they are usually not interested in extracting textures from all objects in the images.

Generally, tracking five points of a known 3D structure along an image sequence is enough to reconstruct the geometric display transformations of the 3D model (four points are enough for a plane). Of the many available automatic tracking methods and approaches, none is absolutely reliable, especially in an environment containing highlights and reflections. A tracking method that can transform each pixel of one frame to a corresponding pixel in the next frame (for example, optical flow) will eliminate the need for a known 3D model, but this is more susceptible to illumination effects than other methods. Another method[9] uses a trifocal tensor for the tracking and is relatively stable under the basic assumptions of this article (described below).

Certain applications inherently require textures from real images—for example, to endow a real object with a personality and orchestrate its motion with animation software.

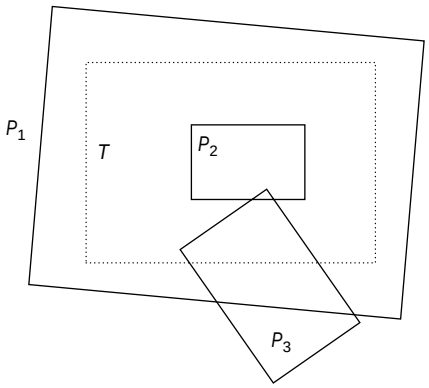
We produced our examples by manually tracking 15 points of a known 3D model on sequences not longer than 16 images. Using 15 points gives us a more reliable result than using five points. A 3D model that is not absolutely accurate (for instance, ignoring round corners of a polygonal object) will generally not affect the tracking but could produce inaccurate local registration in problematic areas.

The method can work either in a batch mode, in which all the images are initially given, or in an incremental mode with images given one by one. Each image is warped onto the desired texture space and inserted into a hierarchical data structure according to its resolution. The color information of each image is fused and distributed at all levels of the hierarchy according to estimated quality. A robust statistical method removes directional illumination effects. This requires a static scene in which only the observer may move.

A note on terminology: Whenever we say "texture," we mean the output of our algorithm, which is actually an image because the input is comprised of images. We call it a texture because its purpose is to be mapped on an object in the course of rendering. The word "image" always refers to the input to our algorithm.

1 Structure of a quadtree node.

```
struct QNode {
    pointer to QNode          child[i]  i ∈ [1, 4]
    RGB                       value
    real                      certainty
    pointer to HighlightInfo  t
}
```



2 The geometric relations between the images and the texture space.

Texture from image sequences

There are two straightforward ways to fuse the texture from all the given images. One way is to average all the values mapped to a certain location in the texture space. Thus each texture pixel's color would be the average of its color as captured in the images. But this can corrupt the quality of the higher resolution images: Consider, for example, what would happen with several distant images of a wall plus a single near image of a picture hanging on that wall. The second way is to choose for each texture pixel the highest resolution image mapped into it. This leads to conservation of noise and other color changes resulting from camera or picture development artifacts.

Our method combines the advantages of the above methods. We use all the available images to determine a value in any texture point using a weighted average that gives priority to the finer details but does not ignore the information contained in lower resolutions. The result is more noise invariant than the second method above and without damage to the fine details produced in the first method.

We describe our method in two stages: first, the simple case, in which the input images are related only by 2D affine transformations, and, second, the general case in which they are related by 3D projective transformations (perspective). We discuss the simpler case first because it is useful in its own right and serves as an introduction to the general case. The mathematical justification for the algorithm appears in the sidebar "Texture Extraction" at the end of this article. We describe the treatment of illumination effects later in the article.

The quadtree representation

A multiresolution texture T is a disjoint cover of the texture space (the unit square) by axis-parallel constant color squares of different dimensions (pixels). In most cases, the dimensions of the pixels are reciprocals of powers of two.

A natural representation for a multiresolution texture is a hierarchical data structure in which a level's resolution is half the resolution of that below it. In a pyramid, each level constitutes a full tiling of the texture space, while a quadtree is a sparse representation in which the levels are not always full. Other variants are possible.

We use a quadtree Q to represent the produced texture. The highest level (root) of Q contains a single node that corresponds to the entire texture space. The next level contains the root's four children, which correspond to the four quadrants of the texture space, and so on. The lowest level nodes correspond to the smallest squares in the multiresolution texture. The quadtree is sparse since its height is less at lower resolution portions of the texture that it represents.

Our representation is not based on differences, for example wavelets or Laplacians (although we generate a Laplacian as an intermediary step). The reason is that the most common query a texture should support is texture mapping, a query best optimized by final color values. Figure 1 shows the information contained within each node of the quadtree Q. The `value` field is the color of the node. The `certainty` field represents the amount of information gathered in the node for weighted average with a new value sampled for that node. The structure *HighlightInfo* is a temporary structure used only to remove directional illumination. It is `null` after eliminating the directional light effects. We describe the structure and its use later in the article.

Pointers to the node's children are stored in `N.child[`i`]`. The tree is as sparse as possible: A new level is opened only where higher resolution texture is required. Texture regions for which only coarse information exists correspond to a shallow subtree.

2D affine transformations

Assume that the textured object lies on a plane and that the image sequence depicting it was generated by an orthographic projection. This situation arises, for example, when the object is relatively distant from the observer or when you are fusing several scanned parts of an image, each in a different resolution, into a single multiresolution image.

Denote the given set of images by P_i, $i = 1, \ldots, n$. Each image is arbitrarily located with respect to the texture: It can be contained within it, contain it, or partially overlap it (see Figure 2).

Assume that the texture space is defined in some standard coordinate system (such as the unit square) and that the 2D affine transformations t_i from the texture space to the image P_i are known for $i = 1, \ldots, n$ (for example, from tracking). We can then compute the texture T in two stages:

1. Construct the tree Q and accumulate the texture information from the images. Determine the resolution for each point in T according to the transformations t_i. Expand the tree Q representing the texture T according to the resolution in every texture point. Add the texture from each image to Q at nodes that correspond to the observed texture resolution in the image.

2. Propagate the information up and down the tree to generate the final multiresolution hierarchical texture. This is necessary to obtain a tree in which each node's value is influenced by all the information stored in the tree.

As described here, the method is a batch process in which all the images are given initially. To express it as an incremental process, with the images given one by one, perform the information propagation step after each image.

```
TreeConstruct(QNode N, Transformation t_i, Image P_i) {
    if t_i(N) is out of P_i's area
        return
    if ResolutionReached(t_i(N))
        ColorsAccumulate(N, t_i, P_i)
        return
    if N is a leaf node
        Expand(N)
    for j ← 1 to 4
        TreeConstruct(N.child[j], t_i)
}
```

3 Construction of the quadtree *Q*.

Constructing the quadtree *Q*. We desire a texture that contains all the information present in the images. Hence the final resolution of any point of the texture T is the maximal resolution at which the point appears in the images P_i where $i = 1, \ldots, n$.

The algorithm starts with a tree Q containing only the root. Using the transformation $t_i : T \rightarrow P_i$, the current resolution of T (the current level in the tree Q) is compared to P_i's resolution. If the texture's resolution is coarser, the tree is further developed until its resolution meets the image's resolution. The texture information from P_i is then added to the proper level in Q using the transformation t_i.

The texture's resolution must not be coarser than the resolution of images mapped to it and should not be much finer (to save storage space). The following steps achieve this goal:

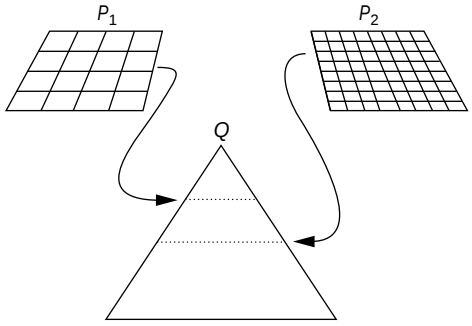
- Project a texture "pixel" (a leaf in tree Q) into the image space P_i using the transformation t_i.
- Examine the resulting quadrangle in the image space to decide whether a local refinement for Q is needed. This decision can be based on any combination of the following criteria: (1) the area of the quadrangle (if it is more than 1, declare that P_i's resolution is finer), (2) the diameter of the projected quadrant (its longest diagonal), and (3) reaching a maximal resolution determined a priori.
- If the above reveals that the image resolution is finer than the local texture resolution, refine the tree (the local texture resolution) by splitting the current leaf.

After developing the tree up to the appropriate resolution according to t_i, we update each node N of that resolution by the value of P_i at $t_i(N)$. The new value of the node is the weighted average of its previous value and the new value from P_i according to the two certainties. The certainty measures the support of a value: The certainty of the new value is 1 while the certainty of $N.value$ is k where $k \leq i$ is the number of values accumulated so far.

Implementation is done by calling the *TreeConstruct* procedure in Figure 3, with Q's root as an argument. The root's certainty is initialized to be 1. The procedure develops Q wherever necessary and accumulates the texture information from the images. The procedure is called for all the input images.

```
ColorsAccumulate(QNode N, Transformation t_i, Image P_i) {
    N.value ← (N.value * N.certainty + Bilinear(t_i(C(N)), P_i)) / (N.certainty + 1)
    N.certainty ← N.certainty + 1
}
```

4 Accumulation of color values in the quadtree *Q*.



5 Two images of different resolutions inserted into *Q* at different levels.

The Boolean function *ResolutionReached* returns True if the current node N is in P_i's resolution using the above criteria, and False otherwise. The *Expand* procedure creates four children for a given leaf with certainties equal to one. The procedure *ColorsAccumulate*, shown in Figure 4, updates N's value and certainty, where $P_i[p]$ is the RGB value of the image P_i at location p. The function *Bilinear* computes a bilinear interpolation of the pixels in the image near which the texture node's center $C(N)$ falls. Note that by construction the size of the texture node in the image is smaller than a pixel, hence its four corners always fall in neighboring image pixels.

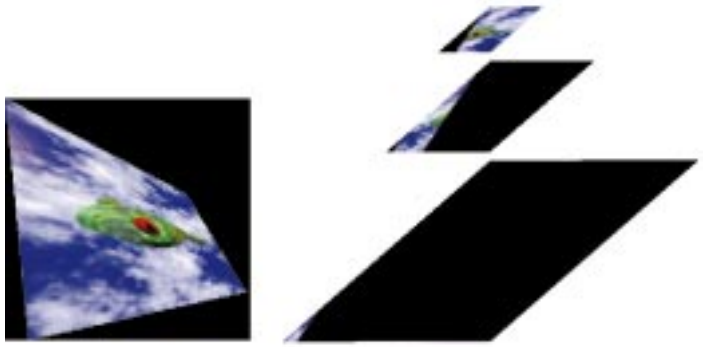
Propagating the information. After the procedure *TreeConstruct* has been called for all of the images P_i, all the color information is stored in the tree, but the levels have not been combined yet (Figure 5). For example, Q's root might contain no value since no image contributed values to its level. We seek to obtain a tree in which each node's value is determined according to all the information stored in the tree. We do this by first propagating the information up from the leaves to the root and then back down to the leaves. The justification appears in the "Texture Extraction" sidebar at the end of this article.

```
TreeUpdateUp(QNode N) {
        RGB ChildrenAverage

        if N is not a leaf node
                for i ← 1 to 4
                        TreeUpdateUp(N.child[i])
                        N.value ← (N.value * N.certainty + N.child[i].value * N.child[i].certainty) / (N.certainty + N.child[i].certainty)
                        N.certainty ← N.certainty + N.child[i].certainty
                ChildrenAverage ← (Σ_{j=0}^{4} N.child[j].value * N.child[j].certainty) / (Σ_{j=0}^{4} N.child[i].certainty)
                for i ← 1 to 4
                        N.child[i].value ← N.child[i].value – ChildrenAverage
}

TreeUpdateDown(QNode N, RGB ParentValue) {
        N.value ← N.value + ParentValue
        if N is not a leaf node
                for i ← 1 to 4
                        TreeUpdateDown(N.child[i], N.value)
}
```

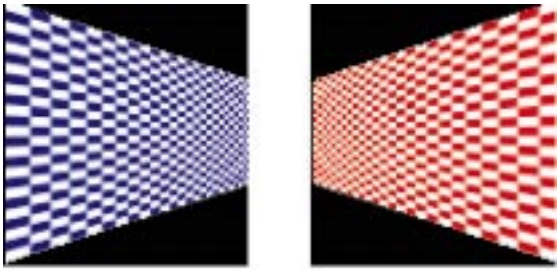
6 Propagation of values up and down the quadtree.



(a) (b)

7 (a) A texture and (b) its corresponding quadtree.

The *TreeUpdateUp* procedure illustrated in Figure 6 propagates the information from the leaves up to the root. *Q*'s root is its argument. The procedure updates each node's value as the average of its previous value and the values of its children weighted according to their certainties. The next step calls *TreeUpdateDown* to propagate the information from the root back to the leaves. It is called with *Q*'s root and zero as arguments. The procedure adds to each node's value that of its parent. Actually, *TreeUpdateUp* generates a sparse Laplacian pyramid representation of the texture, while *TreeUpdateDown* transforms the sparse Laplacian pyramid into a sparse Gaussian pyramid.

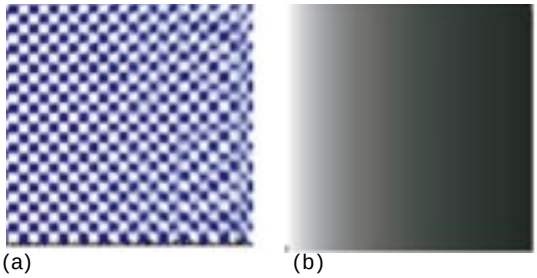


(a) (b)

8 Projection of a uniform net from two different angles. The images are of similar overall resolution and thus appear in the same layer of the texture tree, but the higher resolution appears on (a) the left side of the image and (b) the right side, respectively.

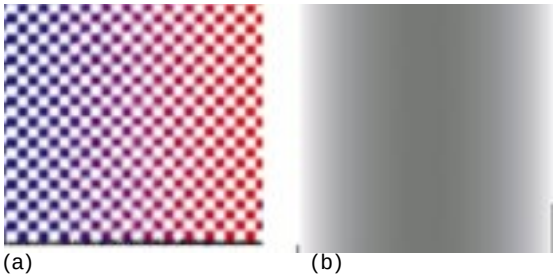
Perspective transformations

Now we can extend the algorithms described so far to the general case: transformation under



(a) (b)

9 (a) The texture resulting from Figure 8a and (b) its certainty map.



(a) (b)

10 (a) The texture resulting from both images in Figure 8 and (b) its certainty map.

perspective projection of any given 3D model. Once again, the input is a set of n images P_i where $i = 1, \ldots, n$ of different resolutions, each covering an arbitrary portion of texture T. The transformations $t_i : T \rightarrow P_i$ from the texture space to the ith image space P_i are now mappings under *perspective* projection. The transformations t_i are easily deduced from the projection of the known 3D model. The problem, again, is to find the resulting texture T.

To do so, we determine the resolution as before, except that an image might have a changing resolution. As an example, Figure 7 demonstrates how to develop a multilevel tree for a single image.

As we accumulate the color data from the images, we must consider the perspective. Two images might have a different certainty even if mapped into the same layer (images will be mapped to different layers only when one resolution at least doubles the other). Figure 8 demonstrates such a case. Both images are mapped to the same layers in the texture tree, but the left side of 8a is of higher quality than the left side of 8b, and the right side of 8b is of higher quality than the right side of 8a. We define quality in terms of the area in the image covered by the tree's node (though other definitions are possible, such as that of Burt and Kolzynski[2]).

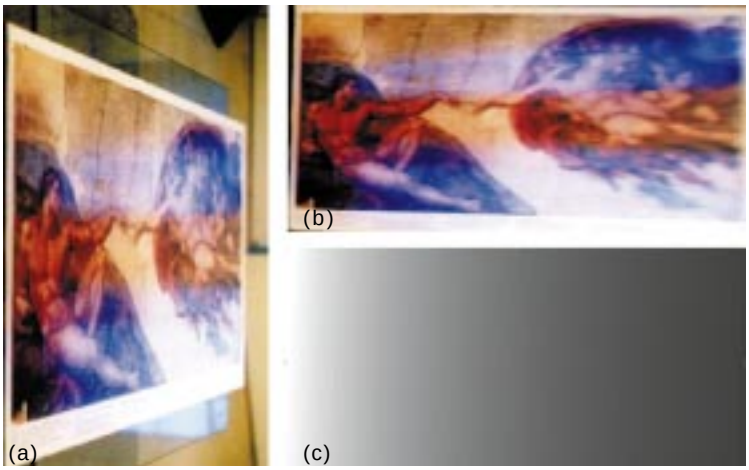
Consider a projection of a uniform net from the texture space into the images (Figure 8). The image's area covered by a single square of the net decreases along with the quality. Figure 9 displays the texture reconstructed using only the image in Figure 8a and its certainty map. In the map, higher certainty values appear brighter.

The texture-accumulating algorithm given earlier can adjust to these new conditions. This happens when we replace the initial certainty that an image sample will be the area covered by projecting a tree node on P_i instead of 1 (see the "Texture Extraction" sidebar for further details). Figure 10 displays the result for the images of Figure 8. Figures 11, 12, and 13 show the same results for the fusion of two real photographs.

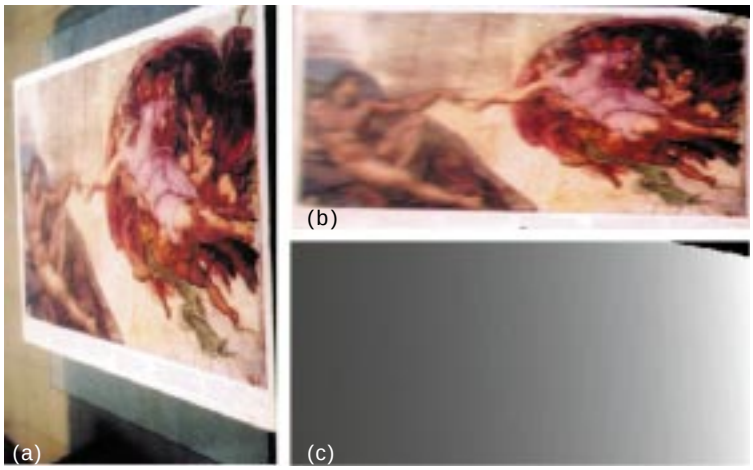
Nonplanar surfaces result in variable resolution in a single image and are treated in the same way (see Figure 14, next page). The only difference is the computation of the transformations t_i.

Removing directional illumination

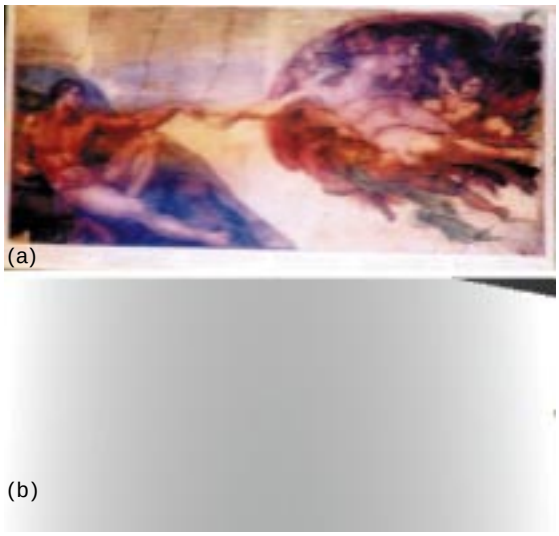
We can remove directional light from images of a surface without any assumptions about the nature of the surface texture. Our technique handles both very rich textures and uniformly colored surfaces.



11 (a) First original image, (b) its mapping into a rectangular texture space, and (c) the certainty map.



12 (a) Second original image, (b) its mapping into a rectangular texture space, and (c) the certainty map.



13 (a) Resulting fusion of the two originals from Figures 11 and 12 and (b) its certainty map.

The algorithm relies on two assumptions: that the scene is static and that each pixel in the texture is free of directional illumination effects in most of the frames. Extracting texture from static objects is at least as important in practice as extracting from moving or deformable objects, so this first assumption does not harm the

14 Four images of a soft drink can (top) and the resulting texture (bottom). Note that this object is not planar.

usability of our results. The second assumption is essential: If a pixel is covered with highlight in most of the frames, it is impossible (even for a human observer) to determine its "real" color value.

Light reflection

Light reflection from a surface is described by the light's color and intensity and the surface's bidirectional reflectance distribution function (BRDF). The BRDF decomposes into three major components (Figure 15):

- Light penetrating a dielectric material and reemerging after subsurface scattering and refraction according to Shafer's model.[7] This is usually called "ideal diffuse" in computer graphics literature and "body reflection" in computer vision literature. Until recently, most considered this component Lambertian, that is, dependent only on the cosine of the angle between the light direction and the surface normal. Oren and Nayar[10] and Wolff[11] showed that the rougher the surface, the stronger its dependence on view direction—it is Lambertian only for smooth surfaces. The amount of light penetrating the surface depends on the angle of incidence and on the material's properties according to its Fresnel coefficient.
- Light that does not penetrate the material, scattering from the surface in the mirrored direction. This is referred to as the "ideal specular" component, and it increases in significance with the smoothness of the surface.
- The directional diffuse component. This is present when the surface is not perfectly smooth and the material's properties enable surface reflection, depending again on the Fresnel coefficient. It can be visualized as a lobe around the mirrored direction, but is not necessarily symmetric around it; it depends on the material and the directionality of the surface's roughness.

We use the following simpler decomposition into two components:

- Directional light—light reflection that depends on the viewing direction. This includes specular and directional diffuse reflection (surface reflection) and only a negligible portion of the ideal diffuse component.
- Indirectional light—light not dependent on the viewing direction. This does not always exist (in metals, for example). It includes most of the body reflection.

Our algorithm can remove the directional illumination effects, including highlights and reflections. The resulting extracted texture is not illumination independent; it depends on the light source's intensities and colors and still contains light gradients and shadows, as in scenes rendered using the radiosity method.

Highlights and reflections

Highlights are the reflections of a light source from an object's surface. Blake and Bülthoff[12] analyzed the behavior of a highlight resulting from a single light source for a static scene with a moving observer. They expressed the highlight disparity between two images as a function of the changes in viewing direction, the local curvature, and the distance of the light source and the observer from the surface.

Motion of highlights in the images depends on the observer and increases when

- the angle between the viewing direction and the surface normal at the point increases,
- the distance to the observer decreases,
- the distance to the light source increases, and
- the local curvature increases.

For nearly planar surfaces, only the first three conditions apply. All hold qualitatively for area light sources, but in this case observers from different viewing directions would see additional shape changes of the highlight.

We treat reflections of other objects on the surface similarly to highlights, since these simply serve as indirect light sources.

The directional light removal algorithm

The directional illumination removal algorithm relies on the motion differences between the scene and the directional light, assuming a static scene and a moving observer. Given sufficient viewing directions in an image sequence, the algorithm extracts the textures in the scene as if rendered using only nondirectional light (for example, using a radiosity algorithm).

The meaning of "sufficient viewing directions" is scene dependent; it depends on all the parameters that determine the size of the highlights in the image. These include roughness, conductivity and other parameters of the materials in the scene, and the projected areas of the light sources on the surfaces. We assume that each part of the object of interest has been seen without significant directional light in most of the images in a given sequence. We further assume that no single angle of view dominates in the sequence. The algorithm uses statistics to find the color independent of viewing direction for each pixel of the texture covering the object.

Typical scenes under ordinary lighting conditions thus require only a few directions for highlight removal. The algorithm operates on the mapped images in two steps:

1. It calculates an approximation of the view-independent color for each texture pixel.
2. It calculates the average of $t_i(T(p))$ for every texture pixel p for which $t_i(T(p))$ is near the above value to decrease the error (recall that t_i maps the texture space into the image space).

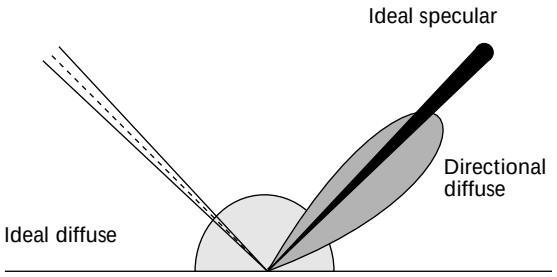
Observe the color histogram that results from following one texture pixel along the image sequence. From the above basic assumption it follows that the histogram will look similar to Figure 16 (shown for a single color band, for simplicity). Most of the histogram concentrates around the value of the view-independent color (the mode of the histogram), and the variations arise from the different amounts of directional illumination that affect the pixel in each image. A small peak appears in the bright area if the texture pixel was covered with highlight in some of the images, and in a few images the texture pixel might get dark values due to insufficient sensitivity of the sensor in the highlight boundaries.

Computing the histogram is very time and space consuming, involving a pass on the image sequence for every texture pixel. But estimating the complete histogram using a few randomly chosen values gives an unstable result, since a value from a texture pixel covered by highlight might deviate the resulting estimate far from the histogram.

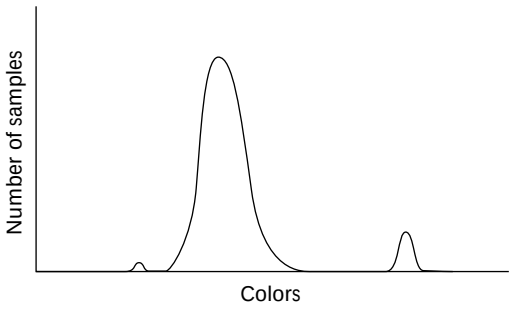
Our algorithm (see Figure 17) estimates the mode (the most frequent value) of the histogram by assuming that a randomly chosen texture pixel from the image sequence is likely to be in the neighborhood of the histogram's mode. It chooses m random values for each texture pixel and calculates their median. The value m is predefined. This median is most likely not far from the color responsible for the mode of the histogram.

This estimate is calculated separately for each texture pixel, so the result is not smooth. Smoothing the texture corrupts its fine details and is difficult to compute in the hierarchical structure. The solution is to calculate, for each texture pixel, the average of its color values along the image sequence, giving us color values that do not deviate much from the median. We calculate the median and the averages separately for R, G, and B. For color value deviation, "much" is 10 percent of the R, G, and B values in our application. In our implementation, m is five. This is large enough to provide a reliable approximation and small enough to store all the required information in memory. We mentioned earlier that the method can work either in batch or incremental mode. In the incremental mode, it initially collects the first m values for each quadtree node.

Figure 18 (on the next page) shows a few frames before and after



15 Components of light reflecting from a surface.



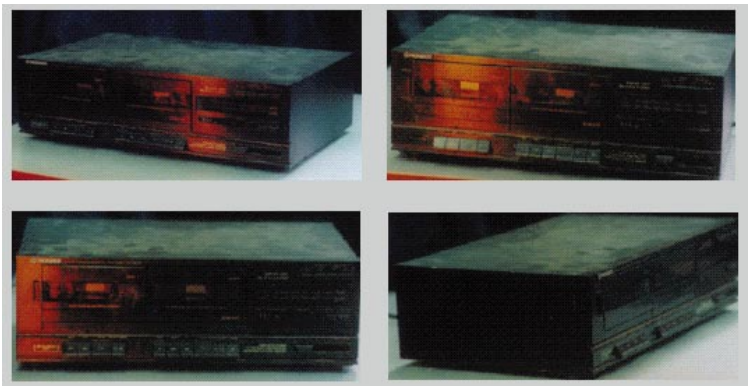
16 A typical histogram of a texture pixel along the image sequence.

directional light removal. The implementation shown in Figure 17 is done by updating the procedure *Colors-Accumulate*. This accumulates the images to their resolution level in the quadtree Q. We assume that the images P_i are given in arbitrary order. Each node in Q is initialized with a *HighlightInfo* temporal storage place, in which m is a predefined value and `counter` is initialized to zero. The first m values from the images that the procedure *ColorsAccumulate* stores in Q are stored in the array $t.a$. When the array contains m values, their median is calculated and stored in $N.t.median$ of the node. For clarity, we update *ColorsAccumulate*, which ignores perspective.

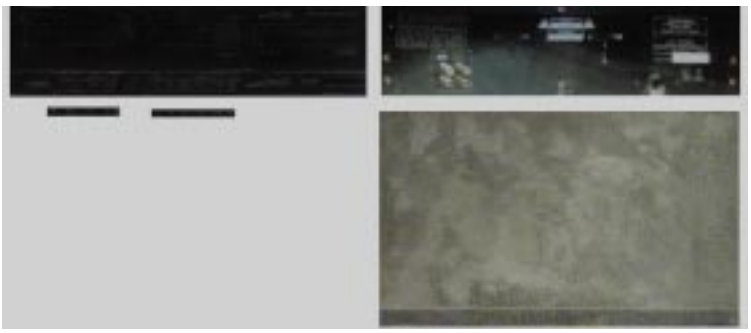
17 The modified color accumulation procedure including highlight removal.

```
struct HighlightInfo {
        RGB                                     a[i]     i ∈ [1, m]
        RGB                                     median
        Integer                                 counter
}
ColorsAccumulate(LQNode N,      Transformation t_i, Image P_i, Integer m) {
        if N.t.counter < m
                N.t.a[N.t.counter] = P_i [t_i (C(N))]
                N.t.counter ← N.t.counter + 1
                return
        if N.t.counter = m
                        N.t.median = MedianCalculate(N.t.a[])
                        for j ← 1 to m
                                if ||N.t.a[j] – N.median|| < threshold
                                                N.value ← (N.value * N.certainty + N.t.a[j]) / (N.certainty + 1)
                                                N.certainty ← N.certainty + 1
                        return
        if || P_i[t_i(C(N))] – N.med || < threshold
                N.val ← (N.val * N.certainty + P_i [t_i (C(N))]) / (N.certainty + 1)
                N.certainty ← N.certainty + 1
}
```

18 Several frames before (left) and after (right) directional light removal.



19 Images given as input to our algorithm.



20 The textures extracted from the images in Figure 19.

light removal techniques that don't use special equipment (such as polarized filters) assume a constant color or a very poor texture. Our technique can extract textures from objects with any known 3D geometric structure, not just planar objects. The resulting texture is stored in an efficient multiresolution data structure.

The quality of the restored texture depends on the accuracy of the given 3D model. Any deviation from the true 3D model will result in an inaccurate mapping for that area and hence an inaccurate texture. It might happen that not all the directional light effects will be totally removed. It happens by violating our assumption (any texture point is not affected by directional light in most of the sequence) or if all the m first samples of a given area were under directional light effects (it is a stochastic method). The first problem can be handled by adding more images to the sequence taken from new angles. The second problem is solved simply by recomputing.

The extracted texture is used to texture-map 3D objects in still and animation image synthesis. Texture mapping is done by an algorithm very similar to Williams' mipmap algorithm.[13] Its only difference from that algorithm is that filtering requires finding the neighbors of a quadtree node, a well-known quadtree operation.

A particularly attractive application of our method produces animation sequences of existing objects endowed with synthetic behavior, as demonstrated in Figures 19 to 21. Figure 19 shows four of sixteen images of a tape deck given as input to our algorithm. Figure 20 shows the resulting extracted textures. Note the realism of the dust on the top part of the tape deck and the removal of the strong yellow highlights and the table's reflections.

Figure 21 shows a frame from an animation sequence in which two compact discs and the tape deck dance to the sound of music. The texture on the table is artificially generated marble.

In the future, we plan to further investigate the initial generation of 3D from input images. We expect to use a combination of mathematical and interactive techniques[14] for this exploration. ■

Conclusion

Our method for computing high-quality multiresolution textures from an image sequence resolves the typical difficulties of extracting textures from image sequences. In particular, it can handle different resolutions and perspective distortions and removal of directional illumination artifacts such as highlights and reflections. The method imposes no restrictions on the computed texture; it can be a constant color texture or a richly colored one. To our knowledge, all other high-

21 **A frame from the "machine dance" animation.**

Texture Extraction

Texture extraction involves finding, for each texture pixel, the estimated color with the highest probability with respect to its colors across all the images. Here we prove that the algorithm for constructing the quadtree Q finds that estimate.

Let T be the reconstructed texture. Let G be a Gaussian pyramid that represents T. Then

$$G^{i-1} = (G^i * g) \downarrow 2$$

where * is the convolution operator, g is the Gaussian filter, and $\downarrow$ is the subsampling operator. We assume, for simplicity, that g is 1 for a node's children and 0 for any other node (so that a node is the average of its four children).

We start with the simple case of 2D transformations and no perspective. The input consists of two images P_1, P_2 containing noise with a distribution $N(0, \sigma)$. The texture T appears in different resolution in each image (but the texture has a uniform resolution within each). Let us now assume, without loss of generality, that P_1's resolution suits level i in G and P_2's resolution suits level $i + 1$.

We can limit the discussion to the five nodes $t_1, \ldots, t_4$ of level $i + 1$ and their parent in the tree, t_p of level i.

Optimal value for t_p

We derive two equations from the two input images:

1. From P_1 we get $t_p = V_p$ where V_p is the pixel in P_1 that corresponds to t_p. The measurement uncertainty is σ^2.
2. From P_2 we get $t_p = \frac{1}{4}(V_1 + V_2 + V_3 + V_4)$ where $V_1, \ldots, V_2$ are P_2's pixels corresponding to the texture pixels $t_1, \ldots, t_4$. The measurement uncertainty is $\frac{1}{4}\sigma^2$

The measurement uncertainty is proportional to the inverse of the number of pixels (sensors) contributing to the measurement. The number of pixels represents the area in P_i and corresponds to the texture covered by t_p.

We find t_p by least-squares estimation, which in this case is equivalent to maximum likelihood estimation:

$$\min_{t_p} \frac{1}{\sigma^2}\left(t_p - V_p\right)^2 + \frac{4}{\sigma^2}\left(t_p - \frac{1}{4}\sum_{i=1}^{4} V_i\right)^2$$

Differentiating with respect to t_p gives us

$$\frac{2}{\sigma^2}\left(t_p - V_p\right) + \frac{8}{\sigma^2}\left(t_p - \frac{1}{4}\sum_{i=1}^{4} V_i = 0\right)$$

so that finally we get

(continued on the next page)

$$t_p = \frac{1}{5} V_p + \frac{4}{5} \frac{\sum_{i=1}^{4} V_i}{4}$$

with the uncertainty $\frac{1}{5}\sigma^2$. Thus, each node's value is the average of its value and its children's value weighted proportional to their area's ratio.

Optimal value for t_1, t_2, t_3, and t_4

In a similar way we obtain the following equation from P_1:

$$V_p = \frac{1}{4} \sum_{i=1}^{4} t_i$$

with uncertainty σ^2 in measuring V_p, and from P_2 we get $t_i = V_i$ where $i = 1 \ldots 4$ with uncertainty σ^2.

The goal is to find

$$\min_{t_j} \left[\frac{1}{\sigma^2} \sum_{i=1}^{4} (t_i - V_i)^2 + \frac{1}{\sigma^2} \left(\frac{\sum_{i=1}^{4} t_i}{4} - V_p \right)^2 \right]$$

where $j = 1, \ldots, 4$

Taking the derivatives according to t_i, we obtain the four new equations

$$t_j = V_j - \frac{1}{4} \left(\frac{\sum_{i=1}^{4} V_i}{4} - V_p \right)$$

where $j = 1, \ldots, 4$

Note that the quadtree Q that was constructed in the article is a sparse Gaussian pyramid; hence, we update level $i + 1$ in Q according to level i above it.

We used propagation of values to obtain the final quadtree. We first propagated values from the leaves to the root to obtain a Laplacian pyramid. In a Laplacian pyramid, each level stores the difference between the equivalent level and the one above in the Gaussian pyramid, so for each node

$$t_i^L = V_i - \frac{\sum_{i=1}^{4} t_i}{4}$$

This results from propagating the values from the leaves to the root:

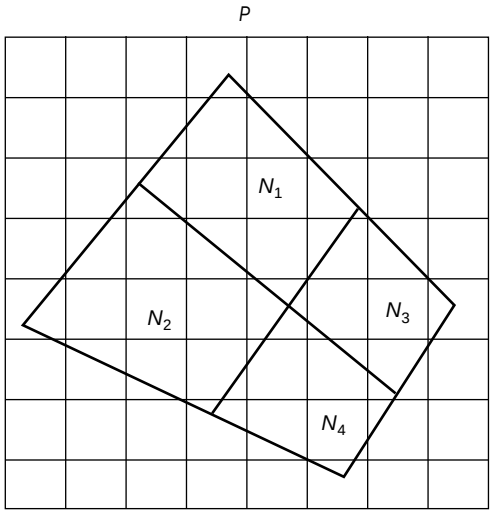
$$t_p = \frac{1}{5} V_p + \frac{4}{5} \frac{\sum_{i=1}^{4} V_i}{4}$$

$$t_i^L = V_i - \frac{\sum_{i=1}^{4} t_i}{4}$$

The final propagation from the root down to the leaves obtains the final result:

$$t_i = t_i^L + t_p$$

The general case of perspective projection preserves this analysis because the certainty of each sampled measurement from an image is proportional to the projected area of the relevant node on the image. The larger the area, the more image pixels (sensors) involved in producing the node's value (see Figure A). The node's value is the average of these pixels, each with noise of distribution of $N(0, \sigma^2)$. If the area of the projection is A, then the certainty is a/σ^2.



A Texture quadtree nodes projected onto the image.

References

1. J.T. Kajiya, "The Rendering Equation," *Computer Graphics* (Proc. Siggraph), Vol. 20, No. 2, Aug. 1986, pp. 143–149.
2. P.J. Burt and R. J. Kolczynski, "Enhanced Image Capture through Fusion," *Fourth Int'l. Conf. on Computer Vision*, IEEE Press, Piscataway, N.J., 1993, pp. 173–182.
3. M. Irani and S. Peleg, "Super Resolution from Image Sequences," *Proc. 10th Int'l. Conf. on Pattern Recognition*, IEEE Computer Society Press, Los Alamitos, Calif., 1990, pp. 115–120.
4. D.F. Berman, J.T. Bartell, and D.H. Salesin, "Multiresolution Painting and Compositing," *Proc. Siggraph 94*, ACM Press, New York, 1994, pp. 85–90.
5. L.B. Wolff, "Scene Understanding from Propagation and Consistency of Polarization-based Constraints," *Proc. Computer Vision and Pattern Recognition*, IEEE Press, Piscataway, N.J., 1994, pp. 1000–1004.
6. G.E. Healey, S.A. Shafer, and L.B. Wolff, *Color,* Jones and Barlett Publishers, London, 1992.
7. S.A. Shafer, "Using Color to Separate Reflection Components," *Color Research and Application*, Vol. 10, No. 4, Winter 1985, pp. 43–51.
8. S.W. Lee and R. Bajcsy, "Detection of Specularity Using Color and Multiple Views," *Image and Vision Computing*, Vol. 10, No. 10, Dec. 1992, pp. 643–653.
9. E. Shilat et al., "Tracking a Rigid Object Along Image Sequences Using a Three-Frame Matching Primitive," Tech. Report, Institute of Computer Science, Hebrew University of Jerusalem, http://www.cs.huji.ac.il/papers/IP/index.html.
10. M. Oren and S. Nayar, "Generalization of the Lambertian Model and Implication for Machine Vision," *Int'l J. of Comp. Vision*, Vol. 14, No. 3, April 1995, pp. 227–251.
11. L.B. Wolf, "Diffuse and Specular Reflection from Dielectric Surfaces," *Image Understanding Workshop*, Morgan Kaufman, San Mateo, Calif., 1993, pp. 1025–1030.
12. A. Blake and H. Bülthoff, "Shape from Specularities: Computation and Psychophysics," *Phil. Trans. Royal Soc. of London*, B 331, 1991, pp. 237–252.
13. L. Williams, "Pyramidal Parametrics," *Computer Graphics* (Proc. Siggraph), Vol. 17, No. 3, July 1983, pp. 1–11.
14. Y. Zakai and A. Rappoport, "Three-Dimensional Modeling and Effects on Still Images, *Computer Graphics Forum* (Proc. Eurographics), Vol. 15, No. 3, 1996, pp. 3-10.

Eyal Ofek *is a doctoral candidate at the Institute of Computer Science at the Hebrew University of Jerusalem, Israel. He has done research in the use of computer vision and computer graphics. He obtained an MS degree in computer science from the Hebrew University of Jerusalem in 1992 and a BS degree in mathematics, physics, and computer science from the Hebrew University in 1987. Research interests include structure from motion, tracking, and rendering.*

Erez Shilat *is a doctoral candidate at the Institute of Computer Science at the Hebrew University of Jerusalem. He obtained his MS degree in computer science from the Hebrew University in 1992, working on continuity of free-form surfaces, and a BS degree, also in computer science, in 1991. Research fields include computer vision and computer graphics.*

Ari Rappoport *is a lecturer at the Institute of Computer Science at the Hebrew University of Jerusalem, Israel. His research interests include geometric modeling, solid modeling, and computer graphics. He obtained his BS in mathematics and computer science and his PhD in computer science from the Hebrew University in 1990.*

Michael Werman *is a senior lecturer at the Institute of Computer Science at the Hebrew University of Jerusalem, Israel, where he earned a PhD in 1986. His current research focuses on computer and robot vision.*

Contact Rappoport at the Institute of Computer Science, The Hebrew University, Jerusalem 91904, Israel, by e-mail at arir@cs.huji.ac.il, or on the Web at http://www.cs.huji.ac.il/~arir.