Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

WUCSE-2002-9

Creating View-dependent Texture Maps

Authors: Cindy Grimm; Michael Kowalski

November 28, 2000

Department of Computer Science And Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: 314-935-6160

Technical report WUCS-2002-9: Creating View-dependent Texture Maps

Cindy Grimm
Department of Computing Science
Washington University in St. Louis

Michael Kowalski
Department of Computer Science
Brown University

Abstract

We present a technique for blending multiple images of an object into a single, view-dependent texture map for that object. This technique can be used for image-based rendering, when the object is known, or for "painting" a view-dependent texture map of an object. The technique provides a structured mechanism for combining images at different resolutions, producing a mip-map like structure with the different levels constructed from different images. The user controls the camera angles for which a given image is valid. The problem of gaps caused by self-occlusion and non-overlapping images is also dealt with. This technique is also suitable for use on an object that will be animated.

Key words: Texture mapping, Surface, Solid, and Object Representations, Splines, Image-based rendering

1 Abstract

2 Introduction

In this paper we present a technique for blending multiple images of an object into a single view-dependent texture map for that object. This problem arises from image-based rendering when the object's geometry is known or can be approximated [2][13]. The images are used to simulate lighting effects produced by camera changes. In this case, much more accurate results can be achieved by blending on the surface of the object, rather than interpolating the images themselves.

The current approach for blending images on the surface of the object creates a texture map for each image and then alpha-blends the texture maps depending upon the current camera orientation. The algorithm in this paper provides a more structured approach that behaves well even in the presence of self-occlusion and images with different resolutions. The algorithm fills in gaps caused by self-occlusion, non-overlapping images, and uneven mapping of the object to image space. Explicit control over the valid range of camera angles for a given image is also provided. The technique incorporates a version of mip-mapping where it is possible to use specific "closer" and "further" images to override the standard blending produced by mip-mapping.

This paper also combines the ideas of image-based rendering with 3D painting [7], allowing the user to "paint" a view-dependent texture map instead of, or in addition to, using captured images. 3D painting of view-dependent texture maps allows the artist to paint effects that come and go based on the camera orientation and depth. To support 3D painting the algorithm employs two intermediate data structures, a view-dependent one and a view-independent one (essentially a base-coat). The user has additional control over how much the base-coat or view-dependent data shows through for a given camera angle. The view-independent data can also be used in the image-based rendering application to provide default colors for the object when no suitable image is available.

The basic outline of the algorithm is as follows. In a pre-processing step the images are blended and resampled into the dependent and independent data structures. At render time, the texture map of the object is colored using the intermediate data "indexed" by the current camera location. The intermediate data is hierarchical, with each layer covering the entire surface in smaller and smaller pieces.

In the next section we discuss related work. In section 4 we define the notions of depth, how to divide the surface up, and how to map from the surface to the image. Following this, section 5 gives a summary of the data structures used. Section 6 describes the algorithm for filling in the intermediate data structures. Section 7 describes how to fill in the texture map from the intermediate data. In section 8 we briefly describe the simple interface used to make the hand-painted examples.

3 Related work

View dependent texture maps for image-based rendering can be found in [2][3][13]. Each image forms a texture map and the texture maps are alpha blended depending upon which ones are closest to the current view. The technique described here expands on this idea to provide more control over the process. Other image-based rendering papers incorporate knowledge of the object to improve interpolation [16][2][5]. The paper [11] changes the texture map of the object based on viewing direction to correct for distortion. With this technique a brick-wall

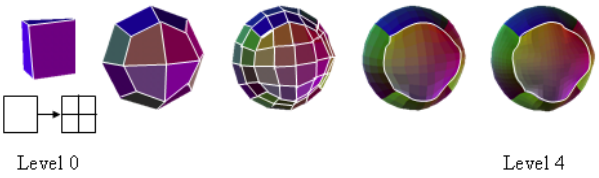

Figure 1: Meshes constructed for each layer of cells. The level zero mesh has six cells, one for each face. The cells are split into four to produce the next layer.

texture will appear 3D. None of the image-based rendering papers deal directly with changes that happen when the camera moves closer or further away from the object, or provide more specific control for hand-painted images. For these types of effects we turn to non-photorealistic rendering.

One of the first papers to indirectly introduce the concept of depth dependent effects was [15] with the concept being refined in [9]. In these papers, the density of the strokes on the surface was determined by the camera's proximity to the object and the image size. A similar notion of depth for procedural textures was introduced in [12]. In [8] the authors introduce a technique for mip-mapping painted or stroke-based textures. We use ideas similar to the above for depth and mip-mapping.

The idea of painting directly on an object to produce a texture map was introduced in [7] and is now available in most commercial systems. This paper extends this notion to hierarchical and view-dependent texture maps.

4 Preliminaries

Before discussing the data structures and algorithm in detail, we first define two concepts which will be used throughout the paper. The first concept is called a *cell*, and is used to divide up the surface into discrete regions. The second concept we define is a version of depth which depends upon both the camera distance and the image size. We then define how we establish the object to image correspondence using cells and depth.

A note on terminology: The term *view* will be used throughout the paper to encapsulate the camera position, image size, and actual image plus an alpha mask. We use the alpha channel of the image to determine what part of the image is to be blended into the intermediate data structures.

4.1 Division of the surface into cells

Informally, a cell is just a portion of the surface. A *layer* of cells is a collections of cells such that every point on the surface is in exactly one cell. We construct several layers of cells so that they form a hierarchy, *i.e.*, each cell in layer i is the union of some of the cells in layer $i+1$. An example of layers zero through four of a spherical surface can be seen in Figure 1.

More formally, we require that the surface be partitionable into relatively evenly sized pieces. By partitioned, we mean that each point on the surface belongs to exactly one cell, where each cell is a mapping from a portion of the surface to a disk in $\Re^2$. This mapping must be 1-1 and onto. Ideally, the surface areas of the cells should be roughly equal. Similarly, their domains (the disks in $\Re^2$) should be the same size[1]. Otherwise, the resolution of the surface will vary. The collection of cells is called a layer. We also place restrictions on the layers, namely that they be hierarchical. The simplest way to achieve this is to define the lowest layer of cells, layer 0, and produce the next layer by subdividing the cells of layer 0. This produces a nested set of layers.

To produce the meshes in Figure 1 we made a mesh with one face per cell for each layer. The vertices are located on the corresponding cell corner on the surface. We call these meshes the *id meshes* because we use them to identify which pixel in an image belongs to which cell [14].

Cell layers for sample surface types

For our implementation we used *manifold surfaces* [6] because the implementation is already structured to support cells; however, subdivision, spline or polygonal surfaces will work as well. The zero layer of cells is created by making a single cell for every vertex chart. We subdivide these cells by splitting each cell into four (see Figure 1). For a given surface we tend to produce a maximum of 3 to 6 layers of cells, depending upon the desired resolution.

A similar scheme works for Catmull-Clark subdivision surfaces, using the quads produced by the first level of subdivision as the layer zero cells. For Loop subdivision surfaces the cells must be split using a triangular cell division scheme. Spline surfaces can use the individual patches as the layer 0 cells.

For arbitrary meshes, the techniques described in [1] or [10] could be modified to produce the layer 0 cells.

4.2 Defining depth

The usual notion of depth is the distance from the camera to the object. In the real world, this corresponds to the object taking up more (or less) of our visual field, with a corresponding gain (or loss) of detail. In our case, the viewer (the OpenGL window) can change the visual resolution of the object in two ways; either by changing the camera distance or by changing the size of the window.

[1] It is possible to account for varying surface area size by adjusting the size of the disks, but this introduces additional complexity.

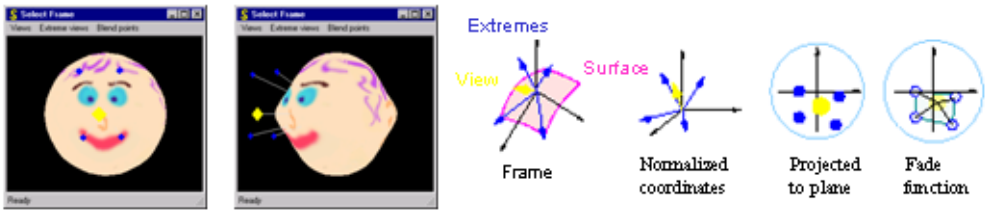

Figure 2: A view direction and its four extreme points. The fade function is constructed by projecting the points onto the plane and building a smooth "hat" function over the resulting polygon by placing the fade curve with its 1 end at the center of the projected view and its 0 end on the boundary of the polygon.

To account for this we use a depth metric based on the number of pixels the object occupies.

Ideally, the depth d of an image is the layer i mesh such that each cell in the mesh covers exactly one pixel. In general we will not have a perfect mapping. Instead, we find the two bracketing layers, i and $i+1$, such that layer i has, on average, fewer cells then pixels and layer $i+1$ has more cells than pixels. The actual depth d lies between i and $i+1$ and is given below.

4.3 Mapping from image to object

We assign a cell from the bracketing id meshes to each pixel in the view's image. We do this using OpenGL. Each face in the id mesh is assigned a unique id. This value is then converted to an RGBA tuple. The mesh is rendered using OpenGL with lighting and anti-aliasing turned off. The RGBA values can then be read out of the image buffer and converted back to their unique ids.

Rather than re-rendering the id image every time we copy colors to the intermediate data, we cache the information in the following form: For each visible cell, we store a list of $(x,y)_i, p_i$ image positions and percentage values. The p_i are normalized to sum to one and represent the percentage each pixel in the list contributes to the final cell color. We also store, for each visible cell, the current color and the current alpha channel mask value, as calculated using the above data.

To minimize gaps caused by the discrete nature of the id image, we actually render the id images at twice the resolution of the view's image. For each pixel in the double-sized id image we add $1/4$ of the corresponding (x,y) pixel in the actual image (recall that we will normalize these percentages).

To chose the bracketing layers we begin with the layer zero id mesh, render, and count the average number of pixels. We continue until the number of average pixels drops below four (recall that our id images are double-sized). At this point we return $d = i - (4 - avg)/4$.

The $(x,y)_i, p_i$ data only needs to be computed once for each bracketing layer. When the image changes, we update the color and mask value assigned to each cell by blending using the cached $(x,y)_i, p_i$ values.

5 Data structures

In this section we define all of the data types

Views: A view consists of an image, camera information, a depth as calculated in section 4.3, and a flag indicating if it is a dependent or independent view.

Extreme points and fade curve: These are used to define the range of camera views for which a given view-dependent image is valid. The extreme points live on the sphere defined by the `from` and `at` points of the camera and indicate the extreme camera ranges (see Figure 2). The fade curve indicates how fast the image should fade out.

Object: The object with the cell structure as described in section 4.1.

Blend points (Optional): A list of camera positions with percentage values indicating the percentage of view-dependent vs. independent data visible from that camera position. These values are interpolated using a nearest neighbor approach (see appendix A).

Independent data: Each cell in each level is assigned an RGB value. These can be stored as portions of a texture map.

Dependent data: Each cell in each level is assigned an nXn array of RGB data representing the hemisphere of color data. Mapping from the hemisphere to the square is illustrated in figures 3 and 2 and the equations given in appendix B.

Texture map: The texture map should have at least the resolution of the highest level of cells. It is also useful to store the 3D surface point for each texture map pixel to speed up the view-dependent calculation.

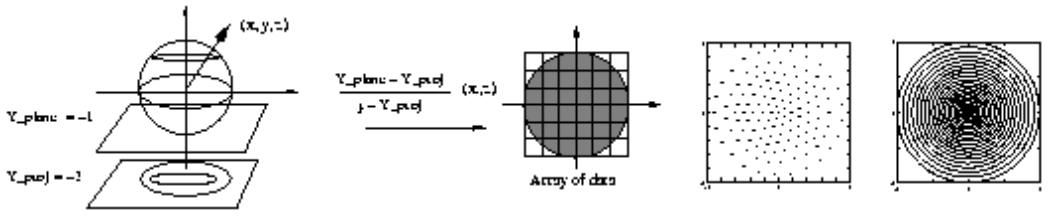

Figure 3: Projecting from the hemisphere of directions to the plane. On the right we show the projection of two different data sets; an evenly distributed set of data points generated from an icosahedron, and concentric rings taken at evenly spaced longitudinal intervals $\theta = \pi/2$ to $\theta = 0$.

6 Intermediate data

We first give a high-level view of the algorithm and then expand on the individual steps.

To fill in the view-independent data we begin at the highest depth level and work our way down to level 0. Each view with a bracketing level at the highest level first fills in all the cells it has data for. These colors are blended where more than one view specifies a color. Any missing pixels are filled in by their neighbors (if there are any nearby neighbors).

To fill in the subsequent levels, we again ask any view at the current level to fill in the cells it has data for. Instead of flood filling, however, we fill in missing data from a filtered version of the previous level.

The view-dependent data is filled-in in a similar manner. Each "pixel" is now a an array of data representing a hemisphere of view directions. The views fill in only the part of the hemisphere indicated by the extreme points. This data is filtered and used in the next lower level. If no data exists at either this level or the previous one then the view-independent data for that cell is used.

6.1 Data filling

For the view-independent data we store a single color for each cell. Note that if all of the independent views have the same depth, then the layers of the independent data would form a mip-map.

For the view-dependent data we store an nXn square of values at each cell representing the hemisphere of directions at that cell. See Figure 3 for a pictorial example of this, and Appendix B for the details of how we convert from the global view directions to the unit square.

For each cell at layer i the view returns a color, blend pair. This color is found as described in Section 4.1. The blend value is the mask value multiplied by a percentage based on the view's depth d. If the view is bracketed by layer i and $i+1$, the depth percentage for layer i is $1-(d-i)$. For layer $i+1$ the depth percentage is $1-(i+1-d)$.

To fill in the intermediate data we simply query each view at the appropriate depth and normalize the results. For instance, a view at depth 2.3 would contribute 0.3 of its color to the cell layer two and 0.7 of its color to the cell layer three. If a second view had a depth of 2.9, the cell layer at level three would be a blend of the two views, $0.3/1.2$ of the first view, and $0.9/1.2$ of the second view. This blending process is illustrated in Figure 4.

For the view-dependent data each cell "pixel" is an nXn array of hemisphere data. The view returns, for each element of the array, its masked color multiplied by the depth percentage and with an additional multiplier, the fade function value for that element. Again, the results are normalized. To produce a smoother blend into the independent data, if the total sum of all the contributing views is $\alpha < 1$, we also blend in $1-\alpha$ of the color of the independent data at that cell.

6.2 Data filtering

If every cell in every layer is covered by some view then we are done. However, in general there will be missing values. These can occur in one of two ways; either a gap was left in a cell layer because the rendered cell mapped to less than $1/4$ pixel, or there are no views in that direction at that level. To address the first problem, we "flood fill" from neighboring pixels. At the highest level we propagate values up to half the level 0 cell size; for lower levels we only propagate to the nearest cell.

To address the missing layer problem, we filter the next higher level and use the filtered values in places with no data. At the highest level missing data is simply set to gray [2].

The view-dependent data is filled-in in a very similar manner except for how to deal with missing data. Each layer has the same size array of hemisphere data for each cell. We can filter these arrays by blending the four arrays of the higher level, lining up the direction values. In addition to filtering the colors we also filter the combined percentages, as returned by the images. When filling in missing data we only use the filtered version if the pre-

[2] It is possible to propagate the data *up* the layers as well by upsampling the images.

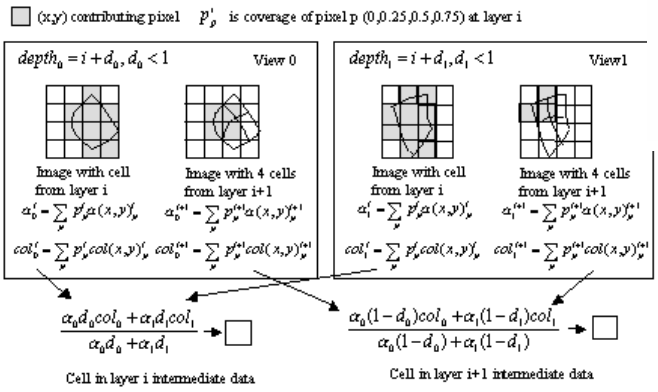

Figure 4: How data is blended from two overlapping views.

vious layer has a non-zero percentage at the cell array value. For the remaining missing data we fill in with the view-independent data for that cell at the same level. The reason we only propagate non-zero filtered data is to prevent "copying" the independent data down through the view-dependent layers.

7 Intermediate data to texture map

At render time we use the intermediate data to fill in the texture map. We index the view-dependent data by the camera orientation and the layers by the new image's depth. We blend between nearby cell values, between the two bracketing layers, and the hemisphere array data. In all of these cases we blend using a linear function.

The reconstruction function we use is a normalized hat function centered on each cell and extending half way into its neighbor cells (see Figure 7). In areas where the cells have a rectangular topology we do not need to normalize and there will be at most four contributing non-zero functions. For manifold surfaces this is the case everywhere except at the vertex chart corners when then number of faces meeting is not four. In this case we simply normalize.

We do this reconstruction for layer i and layer $i+1$ and blend the results according to the new image's depth. Since the cell data is hierarchical, we can re-use most of the computation by multiplying the cell indices by 2.

For the view-dependent data within a cell we use the same reconstruction function on the array of hemisphere data. The point to reconstruct is found by projecting the vector from the current camera to the center of the texture map pixel onto the hemisphere data using the equation in Appendix B. If the point projects to the boundary or beyond we take the closest pixel in the hemisphere array data.

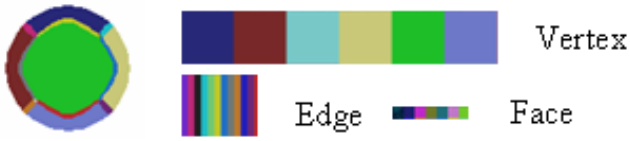

Figure 5: The texture mapped sphere showing the boundaries of the texture map division. On the right are shown the texture maps for the vertex, edge, and face charts.

7.1 Texture map

To fill in the texture map we walk through all the texture map pixels, recomputing their value based on the current depth and vector to the camera. The vector to the camera is taken to be the center of the texture map pixel minus the camera point. This results in a nearly fixed cost per frame (we do not compute values for back-facing pixels). We can also decouple the rendering from the texture map update, enabling interactive camera control even if the texture update is not real-time.

A brief note on texture mapping for manifolds: Since the original paper [6] does not provide details on texture mapping for manifolds, we do so here. Each chart is assigned a portion of the texture map covering the center of the vertex chart, a vertical stripe down the edge chart, and the center of the face chart. These areas correspond to the tessellation, plus a pixel padding around the outside. The texture maps will therefore overlap along the boundaries of the tessellation. Figure 5 shows the texture map for the sphere colored by the texture map areas.

8 A painting interface

The user interface is concerned with creating and editing paintings, not with the actual image creation. The images are created using a paint program or are scanned in. The interaction takes place using three windows. The Results window displays the current object with the current coloring. The object can be colored by coverage (see Figure 6), by just the dependent views, or just the independent views. The Painting window displays the current image. Images are read and written from here. The Select window shows the locations and directions of the views' cameras relative to the object and the extreme points (if any) for the selected view. The blend points and extreme points are manipulated in this window.

9 Results

Figure 8 shows a fish with scales that come and go. The fish runs at 2-5 frames per second on a 800MHz PC. Internal update time is approximately 30 seconds when adding a new view-dependent image. The highest layer

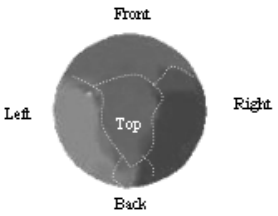

Figure 6: The sphere colored by how much each independent view contributes. The viewing direction is from the top; the regions boundaries are outlined for clarity.

is at depth four and the texture map has $30X30$ pixels per vertex chart. There are 144 vertex charts. The average size for the input images is $270X220$ pixels. There are 9 independent views and 4 dependent ones. There are no depth-dependent effects.

Figure 9 shows a captured teddy bear. Nineteen pictures of a teddy bear (called Stuffy) were taken using a Faro arm, which gives accurate camera positions. The pictures were used both as input images and to create approximate geometry of the object [4]. We constructed a manifold and fit it to the approximate geometry.

Figure 10 shows a vase with a flower pattern. The patterns are view-dependent and only appear when the camera is facing them. There is an additional pattern used to replace the main one when the object is far away. This model has 278 vertex charts and is rendered at level 5. There are three dependent views to specify the pieces of the flower pattern and one view at level 3 to replace the front flowers with a simpler pattern when the vase is at a distance. Note that this object has significant self-occlusion. The independent data consists of 10 views; the shading has been painted in using the independent data.

10 Conclusion

There are several possible methods for creating view-dependent effects on models. We chose to blend and store the data on the object, creating this intermediate storage for the data. The advantages of this approach are that rendering time is constant and the model can be animated. Since the data is on the object, even if we change the geometry of the object, for example, bend it or make it fatter, the view-dependent effects will still occur at the same camera angle relative to the surface normal. There are two disadvantages to this approach. The first is that the storage space is increased over simply storing the images. Second, some detail may be lost or blurred both because of the blending of the data on the object and the reconstruction into the texture map. This second problem is largely addressed by using double-sized id images, which essentially gives us sub-pixel accuracy in the construction of the intermediate data.

Additional problems we address by blending on the object are aliasing and incorrect interpolation of images. The latter problem goes away because we know the location of the object so we can get exact pixel to pixel correspondence between images. The aliasing problem is more subtle and occurs when the re-sampled image size is different than the sampling of the original images, or if the input images have substantially different sampling rates (usually because they were taken at different "depths"). Because we fill in the entire texture map using a linear interpolation of the intermediate data, we will never get gaps or stair steps in the coloring of the object (unless there were no images of that part of the object). The layers of the intermediate data also provide a form of mip-mapping which can be automatic (if the user does not specify any depth-dependent effects) or altered where needed. An example of this is shown in Figure 10.

One issue we do not address completely is control over how the views appear and disappear as the camera changes. The current use of extreme views and a fade curve are useful but not as precise as might be desired. Example alternatives are fading in from the middle out or fading in equally everywhere. One alternative function we have examined is the blend function, which fades all of the visible dependent views in and out as a unit.

A Blend function

The blend function lets the user control how much of the independent versus the dependent is visible. This function is defined over all camera positions p and orientations v and returns a number from 0 to 1 ($B(p,v) \rightarrow [0,1]$). The final color of the texture map is $B(p,v) * C_{indep} + (1 - B(p,v)) * C_{dep}$. The user specifies one or more blend camera positions $(p,v)_i$ and specifies a percentage value $b_i \in [0,1]$ for that point. The system picks a direction clip value c_v and a distance clipping value c_p (currently the average minimum dot product and distance for all blend points). The function is then (all vectors are normalized):

$$s_i = \max(0, \frac{< v, v_i > - c_v}{1 - c_v}) \max(0, 1 - \frac{||p_i - p||}{c_p})$$

$$B(p,v) = \frac{\sum_i b_i s_i}{\sum_i s_i}$$

If $\sum_i s_i$ is zero than $B(p,v)$ returns 0.

B Mapping to the plane

We map a direction vector to the plane by first writing a matrix transform M which takes the frame at the surface point to the Euclidean axes with the normal pointing in

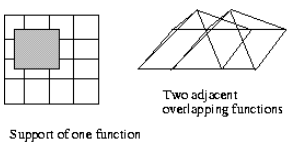

Figure 7: Reconstructing the value at a point on the surface. Resulting point is a linear sum of the four functions overlapping at any given point.

the y direction and the s derivative pointing in the x direction. The direction vector is mapped to the normalized coordinate system using M. The projection to the plane is then:

$$(s,t) = (x,z) * (-1/(y+1))$$

The upper hemisphere maps to a circle of radius of 1. Figure 3 shows the effect of this projection on a set of uniformly distributed points (constructed by sub-dividing an icosahedron) and a set of circles on the sphere formed by evenly incrementing the latitude ($\phi_i = i * \delta$).

C References

[1] Ian Buck, Adam Finkelstein, Charles Jacobs, Allison Klein, David H. Salesin, Joshua Seims, Richard Szeliski, Kentaro Toyama, Emil Praun, and Hugues Hoppe. Lapped textures. *Proceedings of SIGGRAPH 2000*, pages 465–470, July 2000.

[2] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. *Proceedings of SIGGRAPH 96*, pages 11–20, August 1996.

[3] Paul E. Debevec, Yizhou Yu, and George D. Borshukov. Efficient view-dependent image-based rendering with projective texture-mapping. *Eurographics Rendering Workshop 1998*, pages 105–116, June 1998.

[4] Steven J. Gortler. Unpublished work. We appreciate Gortler's having told us about this work.

[5] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. *Proceedings of SIGGRAPH 96*, pages 43–54, August 1996.

[6] Cindy M. Grimm and John F. Hughes. Modeling surfaces of arbitrary topology using manifolds. *Proceedings of SIGGRAPH 95*, pages 359–368, August 1995.

[7] Pat Hanrahan and Paul E. Haeberli. Direct wysiwyg painting and texturing on 3d shapes. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4):215–223, August 1990.

[8] Allison W. Klein, Wilmot W. Li, Michael M. Kazhdan, Wagner T. Correa, Adam Finkelstein, and Thomas A. Funkhouser. Non-photorealistic virtual environments. *Proceedings of SIGGRAPH 2000*, pages 527–534, July 2000.

[9] Michael A. Kowalski, Lee Markosian, J. D. Northrup, Lubomir Bourdev, Ronen Barzel, Loring S. Holden, and John Hughes. Art-based rendering of fur, grass, and trees. *Proceedings of SIGGRAPH 99*, pages 433–438, August 1999.

[10] Fabrice Neyret and Marie-Paule Cani. Pattern-based texturing revisited. *Proceedings of SIGGRAPH 99*, pages 235–242, August 1999.

[11] Manuel Oliveira, Gary Bishop, and David McAllister. Relief texture mapping. *Computer Graphics (Proceedings of SIGGRAPH 2000)*, 34(4):359–368, July 2000.

[12] Ken Perlin and Luiz Velho. Live paint: Painting with procedural multiscale textures. *Proceedings of SIGGRAPH 95*, pages 153–160, August 1995.

[13] Kari Pulli, Michael Cohen, Tom Duchamp, Hugues Hoppe, Linda Shapiro, and Werner Stuetzle. View-based rendering: Visualizing real objects from scanned range and color data. *Eurographics Rendering Workshop 1997*, pages 23–34, June 1997.

[14] Rodney J. Recker, David W. George, and Donald P. Greenberg. Acceleration techniques for progressive refinement radiosity. *1990 Symposium on Interactive 3D Graphics*, 24(2):59–66, March 1990.

[15] Georges Winkenbach and David H. Salesin. Computer-generated pen-and-ink illustration. *Proceedings of SIGGRAPH 94*, pages 91–100, July 1994.

[16] Daniel N. Wood, Daniel I. Azuma, Ken Aldinger, Brian Curless, Tom Duchamp, David H. Salesin, and Werner Stuetzle. Surface light fields for 3d photography. *Proceedings of SIGGRAPH 2000*, pages 287–296, July 2000.

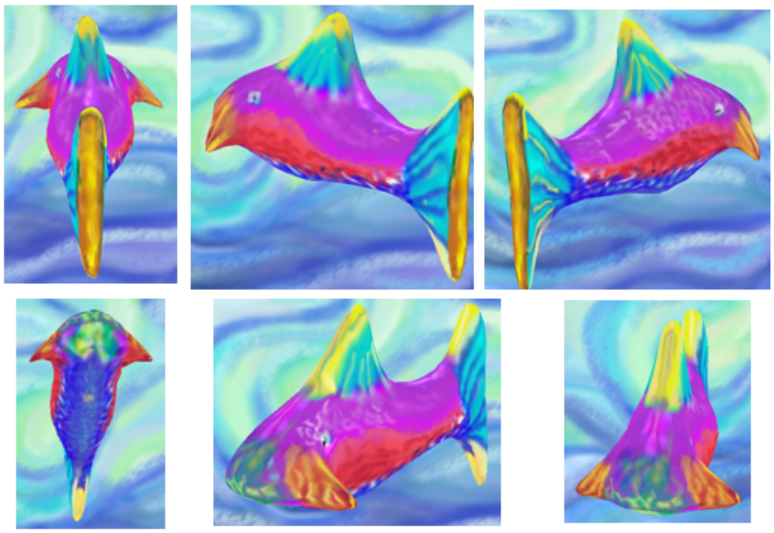

Figure 8: A fish with scales that come and go. Frames are from a video sequence.

Figure 9: A real teddy bear (courtesy of S. Gortler at Harvard Univ.)

Figure 10: A vase with a flower pattern. The side pattern only appears from the side. If the automatic filtering is used, the pattern appears as shown on the bottom when the object is at a distance. On the top, the hand-painted depth effect is shown.