Painting Objects with View-dependent Effects

Cindy M. Grimm, Michael Kowalski

Washington Univ. in St. Louis*

Abstract

We present a system for painting on an object in a view-dependent way. The user creates paintings of the object from various camera positions to produce a texture map which changes depending upon the view direction. The paintings are blended on the object, as opposed to blending the paintings themselves, to ensure that novel views are correctly interpolated. This also produces self-contained models which can then be animated. A painting serves either as a base-coat (a layer which does not change with camera direction) or is blended in depending upon the camera position. If just the basecoat is painted, this system is equivalent to creating a texture map by painting on the object.

There are two parts to the system, a user interface and an internal representation suitable for blending together the paintings on the object. The internal representation is also designed to trade-off speed for accuracy as needed.

Potential uses of this system are artistically created models with control over appearance based on the viewing direction and depth.

Keywords: texture mapping

1 Introduction

In this paper we present a system for painting an object in a view-dependent way. The idea of painting directly on an object in order to create its texture map was first introduced by Hanrahan and Haeberli in [Hanrahan and Haeberli August 1990] and is now widely available in commercial systems. We expand on this idea to let the user also paint effects which come and go depending upon the camera orientation and depth. A simple example is shown in Figure 1, where the object "winks" as the camera moves around the object, and smiles as the camera pulls back.

The purpose of our system is to provide artists with tools to increase their control over the "look" of an object. Currently, effects based on camera position are constructed indirectly. For example, to create a flash off of an object at a certain camera orientation the artist might place a directional light such that its reflection off the object occurs at that orientation. If the object is moved or its reflectance properties changed, the flash will change or disappear altogether. In our system, however, the user would simply "paint" the flash in at the given camera orientation.

*email:cmg@cs.wustl.edu

View-dependent textures also add interesting behaviour to an object without altering its geometry. This behaviour is under the control of the artist, allowing them to focus the attention and add visual detail where needed without cluttering up the scene. For example, adding a complex visual texture to an object makes it more interesting but can be excessive, especially when that object is not the center of attention in a scene. One approach traditional artists use in this case is to "fade out" the texture on parts of the object that do not face the viewer or when the object is far away in a scene. View-dependent textures allow the artist to create a complex, consistent texture *and* specify how much of that texture is visible from a given angle (see Figure 10).

The notion of an object's visual properties changing with the camera position was inspired by image-based rendering. Unlike image-based rendering, however, we are not interested in capturing lighting changes but instead allow the user to create artistic effects that are determined by camera location.

A texture map is not sufficient for capturing view-dependent effects, nor is blending the paintings themselves the correct way to perform the interpolation. Blending the paintings will produce incorrect results especially in the presence of self-occlusion. Since we have the actual geometry of the object, we can blend on the *surface* of the object. This lets us establish exact pixel to pixel correspondences between the paintings. Blending on the surface requires the introduction of a data structure defined on the surface which is suitable for interpolation.

In addition to defining a method for blending on the surface we also demonstrate an interface for painting the object. This interface is concerned with providing feedback, creating and deleting paintings, and controlling the blending of the paintings. We do not provide an interface for creating the actual images; instead, we support importing and exporting of images and let the user use their favorite painting program (or scanner) to create the images for the paintings.

The goal of the system is to retain fidelity to the user's paintings while still providing interactive feedback. To allow this, the resolution of the interpolation data structure (and the texture maps) can be scaled up and down, trading fidelity for speed during construction.

We first discuss previous and related work (Section 2). In Section 3 we introduce concepts which are used throughout the paper. This is followed by a description of the system from the user's point of view (Section 4). Section 5 describes the data format and how it is initialized from the user-supplied data. Finally, we close with a results section.

2 Related work

This work is an extension of ideas presented in [Hanrahan and Haeberli August 1990], whose key idea was to hide the parameterization of the texture map from the user by letting the user paint directly on the object.

The view-dependent portion of this work is closely related to image-based rendering, where different views are interpolated to simulate changing light effects. There are several image-based rendering papers which incorporate knowledge of the object to improve image interpolation [Debevec et al. June 1998][Pulli et al.

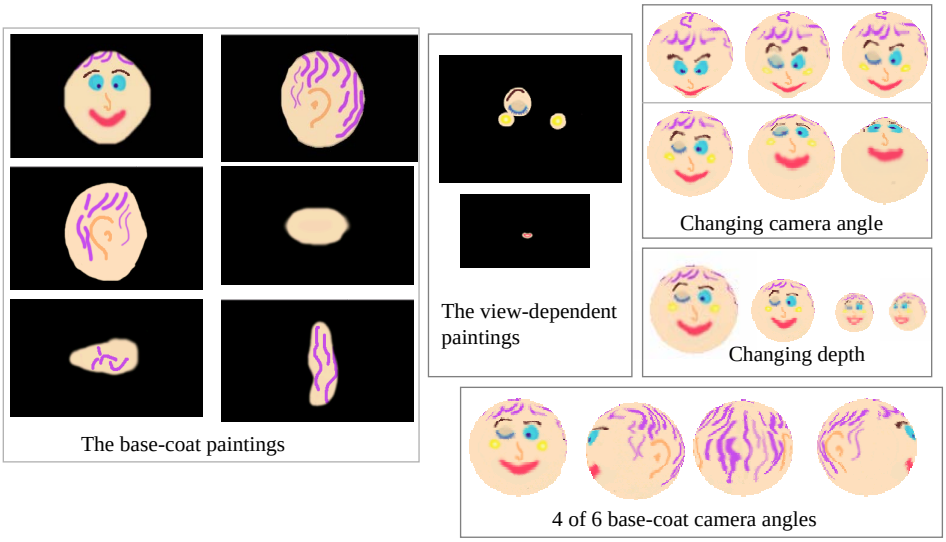


Figure 1: A face which "winks" as the camera moves from left to right or up and down. The mouth opens as the camera is pulled back. All images are shown half size.

June 1997] [Wood et al. July 2000]. Like these papers, we interpolate on the object; however, unlike the first two papers, we store the interpolation data on the object. The paper [Oliveira et al. July 2000] changes the texture map of the object based on viewing direction to correct for distortion. With this technique a brick-wall texture will appear 3D. None of the image-based rendering papers are concerned with changes that happen when the camera moves closer or further away from the object. 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 [Winkenbach and Salesin July 1994] with the concept being refined in [Kowalski et al. August 1999]. 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 [Perlin and Velho August 1995]. We use a notion similar to the ones described above to allow the user to paint effects that depend upon the number of pixels the object occupies.

There are many papers which allow the user to adjust for distortion in the texture map, for example [Maillot et al. August 1993] [Arad 1997]. Other techniques provide a texture map for an object which does not have a natural parameterization [Buck et al. July 2000] [Neyret and Cani August 1999] [Bloomenthal and Wyvill March 1990]. In [Hutchinson et al. August 1996], the user draws 2D planar graphs into the domain of the surfaces. These systems are all useful for creating texture maps but do not create view-dependent ones.

3 Preliminaries

Before discussing the user interface and implementation 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.

A note on terminology: The term *painting* will be used throughout the paper to encapsulate the camera position, image size, and actual image plus an alpha mask. In image-based rendering literature this combination is usually referred to as a "view". Internally we use the alpha channel of the image to determine what part of the image is actually used in the painting (the mask) but from an interface point of view the mask is read and written separately as a grey scale image.

3.1 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 2.

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 2 we make a mesh for each layer with one face per cell. The vertices are located on the corresponding cell corner on the surface. Note that the lowest layer mesh will be, in general, a poor fit with the original surface.

We call these meshes *id meshes* because we use them to identify which pixel in an image belongs to which cell [Recker et al. 1990]. We explain how to do this in detail in Section 5.1.1.

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

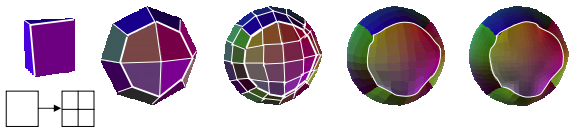


Figure 2: Meshes constructed for each layer of cells. The level zero mesh has six cells, one for each face. The cells are split into four to produce the next layer.

3.1.1 Cell layers for sample surface types

For our implementation we used *manifold surfaces* [Grimm and Hughes August 1995]. 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 2). 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 can be used for Catmull-Clark subdivision surfaces by 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 [Buck et al. July 2000] or [Neyret and Cani August 1999] could be modified to produce the layer 0 cells.

3.2 Depth dependent effects

In this section we discuss the concept of "depth" in our application. 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. To account for this, we use a depth metric which corresponds to the number of pixels the object occupies.

To compute the depth we count the average number of pixels per cell for the front facing, non-occluded cells. We could compute this analytically, by casting rays through the corners of cells and determining the area on the screen of the corresponding projected polygon. This could get costly, however, so instead we use a simpler method of finding the depth. We simply render the id mesh and count the average number of pixels per cell. We describe this in more detail in Section 5.1.1.

We only compute a painting's depth level at its construction or when the maximum depth changes, so this operation need not be real-time. However, there is another use of this computation — determining the depth of the display view. We can calculate the appropriate depth by temporarily creating a new painting with the current camera parameters. Fortunately, we really only need to recompute this value when the camera depth changes or the window size is changed, provided the object is centered in the view port. For high quality, off-line rendering, we can compute this value at every step.

4 The user's view

As mentioned before, the user interface is concerned with creating and editing paintings, not with the actual image creation. The images can be created using a paint program or scanned in. The interaction takes place using three windows (see Figure 3):

- Results window. The current object with the current coloring.
- Painting window. The current painting.
- Select window. The locations and directions of the paintings' cameras relative to the object.

A typical user interaction session might go like this. The user positions the camera in the Results window to the desired view direction and image size. They then request a new painting, specifying either base-coat or view-dependent. (Alternatively, the user picks one of the current paintings to edit.) The user than sets or alters the image and the mask of the painting. Various tools are provided to determine where the other paintings overlap and to give clues as to where the different parts of the object project to. The latter is useful for determining occlusion of one part of the object by another part. After the user is satisfied with the painting, the data structures are updated and the result shown in the Results window.

If the painting is a view-dependent one, the user then adjusts the range of camera positions for which the painting is valid. The boundary of the valid region is determined by a set of extreme camera positions which surround the view (see Figure 5). The user creates new extreme positions by positioning the camera of the Select window and requesting a new point. Existing extreme points can also be moved interactively by grabbing and pulling on them (see Section 4.3).

The user can also adjust the percentage of base-coat which shows through. This control is independent of the extreme camera positions. This is accomplished by choosing particular camera positions and giving the percentage for that camera position (see Figure 11 for an example use). The system automatically interpolates between these sample points (see Section 6).

We now discuss the user interaction in more detail.

4.1 Results window

This window is primarily for viewing the object, although it also serves as the mechanism for inputing camera positions for new paintings. There are two different modes for viewing the object. The first mode updates the texture map only when requested, the second mode updates the texture map with the new camera position every time the camera moves. The first mode is useful when painting the base-coat or for checking specific camera positions when the resolution is too high to allow interactive rates.

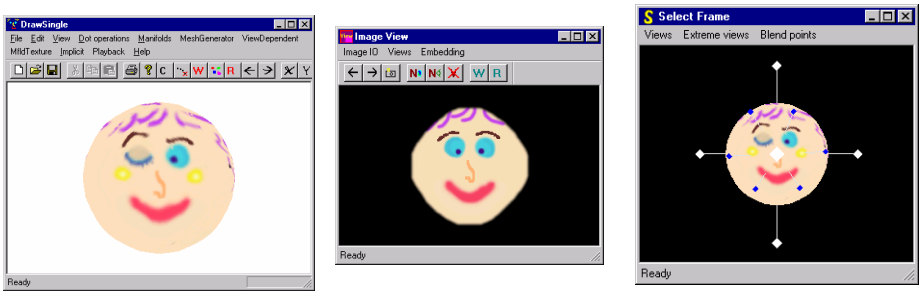
When making a new painting, the camera position and window size of the Results window dictates the camera position and image size of the new painting.

4.2 Painting window

From this window the user imports and exports images. The window always shows the masked image of the selected painting (if there is one). We read and write images and masks separately (although the mask is stored internally as the alpha channel of the image). This allows the user to keep a complete picture of the object for reference purposes, even if only a small part of the image will be used (as specified by the mask). When a new painting is created, its image is initialized with the current rendering (as shown in the Results window) and the mask is initialized to be white where the object is and black everywhere else.

At any time the user can grab an image of the following (using the painting's camera parameters):

- A picture of the object's state as currently shown in the Results window.
- A picture of the id mesh image.



Results window (3D) Painting window (2D) Select window (3D)

Figure 3: The three windows used for interaction. Left is the Results window, middle is the Painting window, and right is the Select window. The cameras of the Results window and the Select Window are independent.

- A picture of the current image (without the mask).
- A picture of the current mask as a grey scale image.

There are also some useful ways to color the object in the Results window:

- Color the object only with the base-coat.
- Color the object according to how much each painting influences the color. Each painting is assigned a unique color and these values are blended on the object to produce the texture map (see Figure 4).

The user can also snap the Results window's camera to be the current painting's camera or its extreme points.

Finally, the user can also set the desired level of fidelity. There are three numbers which control the update speed of the internal data structures, and one number for the rendering update speed. The first update number is the maximum depth (layer) of the base-coat, which determines the amount of time it takes to update the base-coat. There are two numbers for the view-dependent paintings. The first number is the maximum depth for the view-dependent paintings. The second number is the resolution in the directional domain or how many pixels to assign for the hemisphere data for a single cell. The final number is the resolution of the texture map itself in number of pixels per layer 0 cell. The higher the resolution, the longer it takes to update when the camera changes, but the better the texture map.

For all of these numbers, the system provides the number needed to achieve complete fidelity.

4.3 Select window

This window shows the relationships between the camera positions of the various paintings, the extents of the view-dependent paintings, and how much the base-coat shows through. There are three major modes of interaction:

- Change the selected painting. This consists of simply clicking on the dot corresponding to the desired painting. The Painting window will be updated to show the selected painting.
- For a view-dependent painting, set or alter the extreme valid camera positions and how quickly the painting fades out.
- Change how much the base-coat shows through for a given camera position.

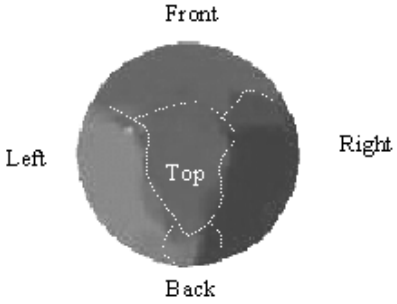
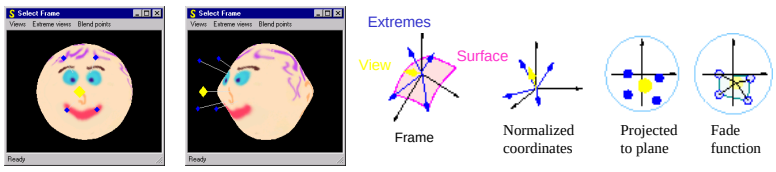


Figure 4: The sphere colored by how much each base-coat painting contributes. The viewing direction is from the top; the regions boundaries are outlined for clarity.

The arrow direction is determined by the `from` and `at` points of the painting's camera. The distance out along the arrow is determined by the painting's depth. The lower the depth, the longer the arrow. The camera of either the Result or the Select window can be snapped to any selected painting or extreme position.

If the selected painting is a view-dependent one, then the painting's extreme camera positions are also shown. These extreme points live on the sphere defined by the `from` and `at` points of the camera. The points can be moved on that sphere by grabbing and pulling on them. A new extreme point is created by moving the camera to the desired extreme position and pressing the "Add Extreme Painting" button. Points can also be deleted, although at least three must remain. There is also a function which determines how quickly the painting fades out (see Section 5.1.2). The function is set using two sliders; the first adjusts when the fade starts, the second adjusts how quickly the fade happens.

There is a function which returns, for any camera position and direction, the amount the base-coat should show through (see Section 6). This is a separate control from the extreme points. Normally, this function simply returns zero, and blending between the base-coat and extreme views depends only on the defined extreme points. To create a more complicated effect, such as "rippling" between the base-coat and a view-dependent painting, the user can optionally set the blend value to a value other than zero (see Figure 11 for an example). The blend value function simply interpolates between a set of sample points. To set the sample points, the user positions the camera to the desired camera location and requests a new sample point. The blend value is adjusted using a slider. Existing sample points can be deleted, moved, and have their values



Four extreme positions around a dependent view Projecting to the plane to produce fade function

Figure 5: 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.

adjusted.

5 Internals

There are three levels of data structures:

- The actual data the user inputs, which we call a painting, consisting of images, masks, and camera positions.
- The intermediate data which has two parts to it, the base-coat data and the view-dependent data. This data is constructed by blending the user's paintings. The resolution of this data determines the update time when the user changes the images or extreme points.
- The texture map itself. This is updated from the intermediate data. The update time is determined by the number of pixels in the texture map, not by the resolution of the intermediate data.

The next sections expand on these three data types, and also describe how data is stored and blended from level to level.

5.1 User's data

The user's data is called a painting and consists of a known camera position, image size, and a masked image. The camera position is chosen by setting the camera and image size on the Results window. The image has an optional mask, stored in the alpha channel. (We provide methods for reading and writing the mask as a grey scale image.)

Each painting has an ideal depth, chosen so that the corresponding id mesh, when rendered using the camera parameters and image size, has roughly one pixel per cell.

Each painting is flagged as either being a base-coat or a view-dependent painting. Users can toggle the type.

View-dependent paintings have additional data indicating the range of camera positions the painting is valid for. These are stored as 3D points on the sphere formed by the `from` and `at` points of the camera. There must be at least three of them surrounding the camera position. See Figure 5 for an example. The painting also has a curve indicating the rate of fall-off.

5.1.1 Depth and the id mesh

In this section we discuss how to assign pixels in the painting's image to the cells of an id mesh at a given level. Each face in the id mesh is assigned a unique id, which is then converted to an RGBA value. 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 mask value, as calculated using this 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 painting'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).

The $(x,y)_i, p_i$ data only needs to be updated when the maximum allowable depth changes. When the image or its mask changes, we update the color and mask value assigned to each cell by blending using the $(x,y)_i, p_i$ values.

The last thing we need to do is define a painting's "ideal depth". The ideal depth for an image would be a partition of the object's surface such that each cell mapped exactly to a single pixel. We could conceivably compute such a partitioning but it would be expensive; instead, we pick the two adjacent layers which bracket the ideal one. The ideal depth is a real number lying somewhere between these two layers.

To chose the bracketing layers we begin with the layer zero id mesh, render, and count the average number of pixels. We continue until we have reached the highest depth level or until the number of average pixels drops below four (recall that our id images are double-sized). At this point we return the current depth minus $(4 - avg)/4$. If we have reached the highest depth level, we return that and the two bracketing layers will be the same.

Since we use two different id meshes we need to cache the $(x,y), p$ data for both of these bracketing layers.

5.1.2 Extreme camera positions

For paintings which have view-dependent data we need a way to define what part of the hemisphere of directions the data is good for. For this we use the extreme points and the fall-off curve. Again, this data can be pre-computed and stored, and only needs to be updated when the extreme points or the fall-off curve change, not when the image changes.

Determining the array of cell color values for the view-dependent paintings is done by "splatting" the pixel color onto some portion of the hemisphere (see Figure 5 and 6). To figure out the region of the hemisphere we need to update, we first project the painting's camera and extreme points onto the plane. We then form a polygon from the projected extreme values, sorting them radially around the projected camera. Finally, we construct a fall-off function by taking the fall-off curve and scaling and rotating it so it is one at the projected camera, and goes to zero by the boundary of the polygon. The exact projection function is given in Appendix A.

The final mask value for a given hemisphere pixel in a given cell is the product of the mask value for the cell, and the fade function at that pixel (see the following section for how the hemisphere data is stored in the cell).

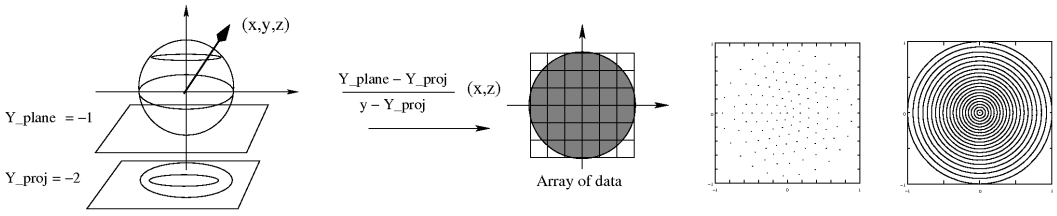


Figure 6: Projecting from the hemisphere of directions to the plane. On the right we show the projection of two different data sets; an evenly distributed set of data points generated from an icosahedron, and concentric rings taken at evenly spaced longitudinal intervals $\theta = \pi/2$ to $\theta = 0$.

5.2 Intermediate data construction

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

The intermediate data is a hierarchical structure consisting of arrays of color data. This hierarchy corresponds to the cell layer hierarchy. For a given level in the hierarchy there is one "pixel" for each cell. For the base-coat data, each "pixel" is a single color. For the view-dependent data the "pixel" is an array of colors corresponding to the hemisphere of colors projected to the plane.

To fill in the base-coat data we begin at the highest depth level and work our way down to level 0. Each painting 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 painting 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 painting 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 paintings 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 base-coat data for that cell is used.

5.2.1 Data filling

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

For the view-dependent data we store an array of values at each cell representing the hemisphere of directions at that cell. See Figure 6 for a pictorial example of this, and Appendix A for the details of how we convert from the global view directions to the unit square.

We have already discussed how to assign colors to cell ids for a single painting. To fill in the intermediate data we simply query each painting at the appropriate depth and normalize the results. For instance, a painting 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 painting had a depth of 2.9, the cell layer at level three would be a blend of the two paintings, $0.3/1.2$ of the first painting, and $0.9/1.2$ of the second painting.

For each cell at layer d the painting returns a color, percentage pair. This color is found as described in Section 5.1.1. The percentage is the mask value multiplied by a percentage based on the ideal depth d. If the painting is bracketed by layer i and $i+1$, the percentage for layer i is $1-(d-i)$. For layer $i+1$ the percentage is $1-(i+1-d)$.

This blending process is illustrated in Figure 7.

For the view-dependent data each cell "pixel" is an array of hemisphere data. The painting 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 values are normalized. To produce a smoother blend into the base-coat data, if the total sum of all the contributing paintings is $\alpha < 1$, we also blend in $1-\alpha$ of the color of the base-coat at that cell.

5.2.2 Filtering

If every cell in every cell layer is covered by some painting 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 paintings 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 paintings. When filling in missing data we only use the filtered version if the previous layer has a non-zero percentage at the cell array value. For the remaining missing data we fill in with the base-coat data for that cell at the same depth. (If the base-coat maximum depth is less than the current view-dependent depth, we take the highest level available.) The reason we only propagate non-zero filtered data is to prevent "copying" the highest level base-coat data down through the view-dependent layers.

5.3 Intermediate data to texture map

In this section we discuss assigning colors to texture map pixels. If the texture map is partitioned exactly the same as a given layer and we are at that layer's depth we could just copy the intermediate data values. In general this will not be the case and we need to blend between nearby cell values. We also need to blend between the two bracketing layers and the hemisphere array data. In all these cases we blend using a linear function.

The reconstruction function we use within a cell layer and for the hemisphere data is a normalized hat function centered on each

[2] It is possible to propagate the data *up* the layers as well, which would be useful if the artist wanted to add detail at a higher level but keep the current base-coat.

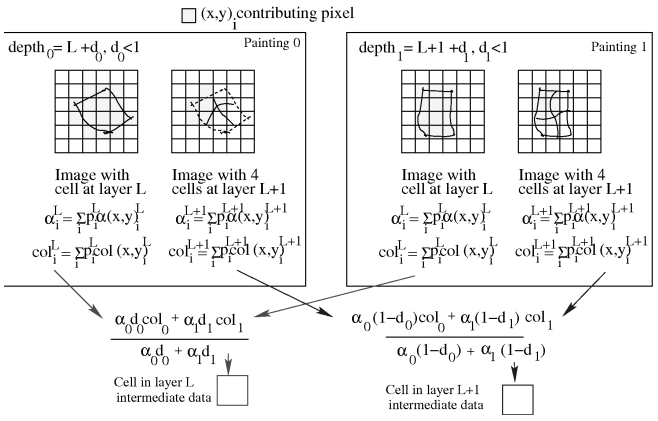


Figure 7: How data is blended from two overlapping paintings.

cell and extending half way into its neighbor cells (see Figure 8). 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 Results window's current depth. Since the cell data is hierarchical, we can re-use most of the computation, simply multiplying the cell indices by 2.

For the view-dependent data within a cell we use the same reconstruction function but applied to 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 A. If the point projects to the boundary or beyond we take the closest pixel in the hemisphere array data.

5.4 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. Ideally, the number of pixels in the texture map should reflect the "depth" of the object, as described in Section 3.2. However, we currently cache the surface normal and blend info for each texture map pixel, so altering the number of pixels is expensive. Instead, we simply pick a high enough resolution for the current stage of editing and let OpenGL's mip-mapping adjust the texture map resolution for us.

A brief note on texture mapping for manifolds: Since the original paper 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 7 shows the texture map for the sphere colored by the texture map areas.

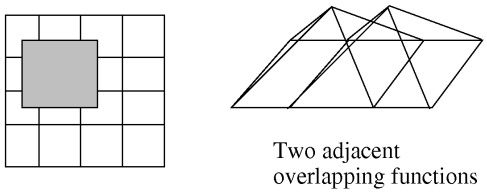


Figure 8: Reconstructing the value at a point on the surface. Resulting point is a linear sum of the four functions overlapping at any given point.

6 Blending the base-coat

In this section we describe an additional function which lets the user control how much of the base coat shows through independent of the view-dependent paintings. This is a function defined over all camera positions p and orientations v which returns a number from 0 to 1 ($B(p,v) \rightarrow [0,1]$). The final color of the texture map pixels is $B(p,v) * C_{base-coat} + (1 - B(p,v)) * C_{dep}$. Normally this functions is 0 everywhere. To alter this function, the user creates one or more blend camera points $(p,v)_i$ and specifies a value $b_i \in [0,1]$ for that point. We then specify 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.

A 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 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 6 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 ($\phi_i = i * \delta$).

References

ARAD, M. 1997. Isometric texture mapping for free-form surfaces. *Computer Graphics Forum 16*, 5, 247–256.

BLOOMENTHAL, J., AND WYVILL, B. March 1990. Interactive techniques for implicit modeling. *1990 Symposium on Interactive 3D Graphics 24*, 2, 109–116.

BUCK, I., FINKELSTEIN, A., JACOBS, C., KLEIN, A., SALESIN, D. H., SEIMS, J., SZELISKI, R., TOYAMA, K., PRAUN, E., AND HOPPE, H. July 2000. Lapped textures. *Proceedings of SIGGRAPH 2000*, 465–470.

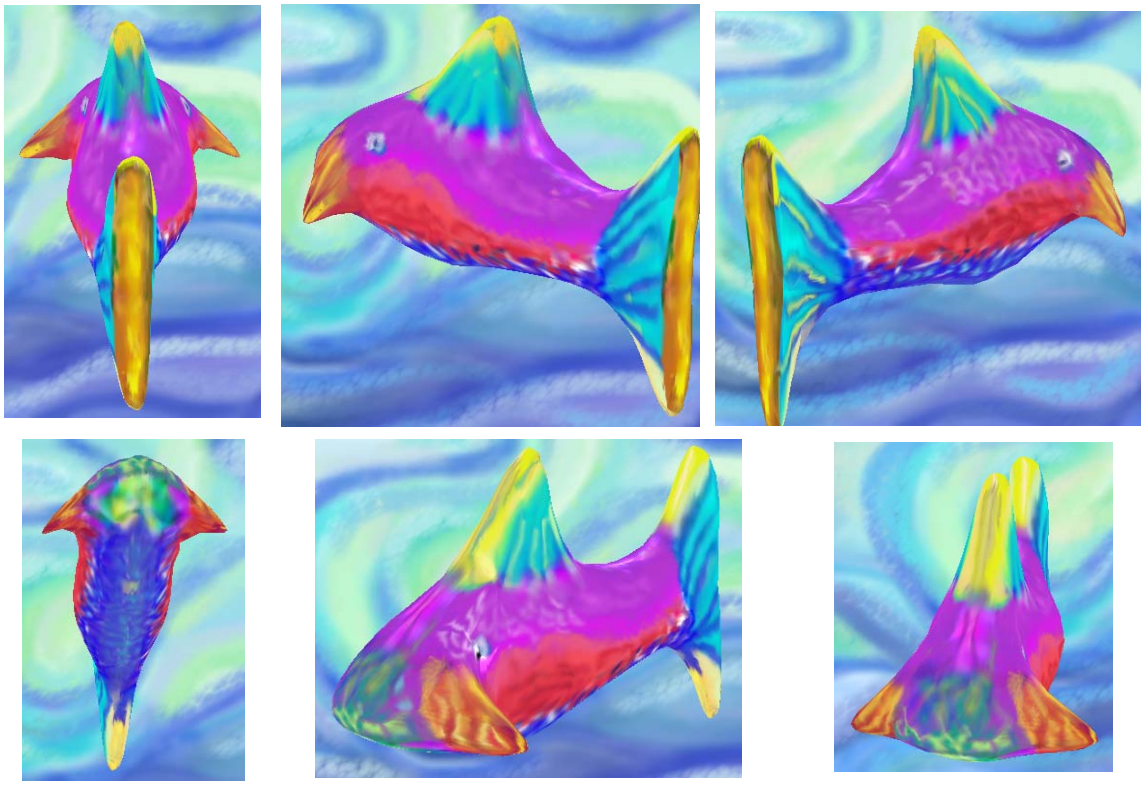


Figure 9: A fish with scales that come and go. Frames are from a video sequence.

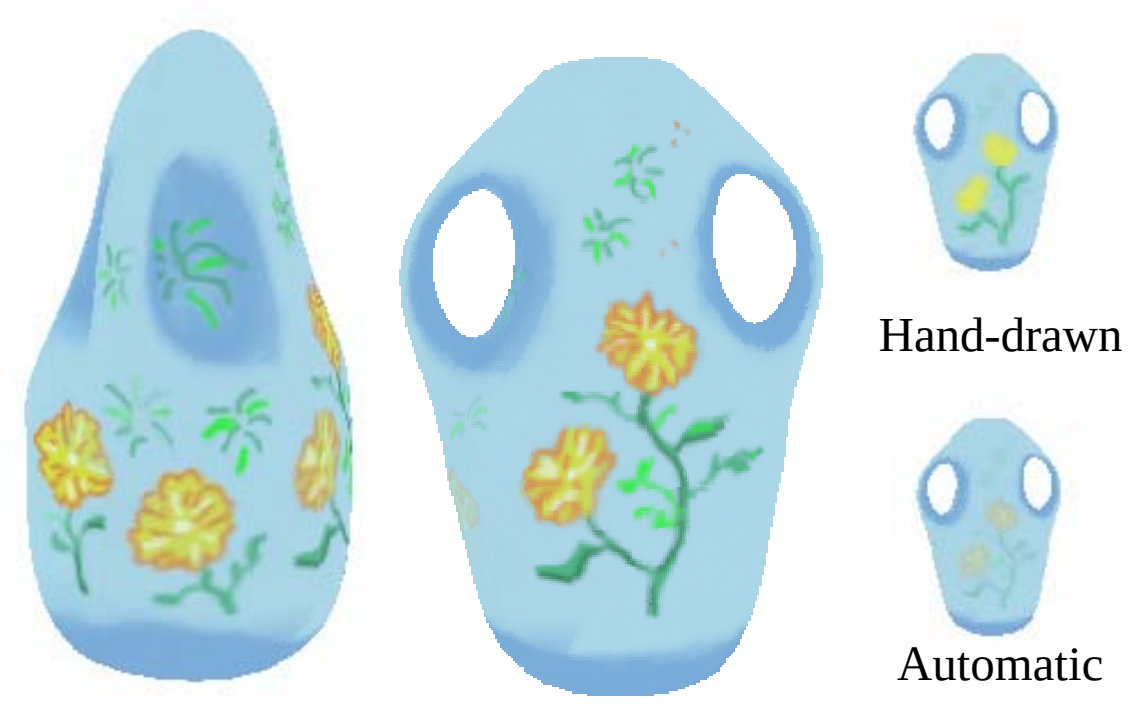


Figure 10: A vase with a flower pattern. The side pattern only appears from the side. If the automatic filtering is used, the pattern appears as shown on the bottom when the object is at a distance. On the top, the hand-painted depth effect is shown.

DEBEVEC, P. E., TAYLOR, C. J., AND MALIK, J. 1996. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. *Proceedings of SIGGRAPH 96* (August), 11–20.

DEBEVEC, P. E., YU, Y., AND BORSHUKOV, G. D. June 1998. Efficient view-dependent image-based rendering with projective texture-mapping. *Eurographics Rendering Workshop 1998*, 105–116.

GORTLER, S. J. Unpublished work. We appreciate Gortler's having told us about this work.

GORTLER, S. J., GRZESZCZUK, R., SZELISKI, R., AND COHEN, M. F. 1996. The lumigraph. *Proceedings of SIGGRAPH 96* (August), 43–54.

GRIMM, C. M., AND HUGHES, J. F. August 1995. Modeling surfaces of arbitrary topology using manifolds. *Proceedings of SIGGRAPH 95*, 359–368.

HANRAHAN, P., AND HAEBERLI, P. E. August 1990. Direct wysiwyg painting and texturing on 3d shapes. *Computer Graphics (Proceedings of SIGGRAPH 90) 24*, 4, 215–223.

HUTCHINSON, D., LIN, F., AND HEWITT, T. August 1996. Surface graph sketching. *Computer Graphics Forum 15*, 3, 301–310.

KLEIN, A. W., LI, W. W., KAZHDAN, M. M., CORREA, W. T., FINKELSTEIN, A., AND FUNKHOUSER, T. A. 2000. Non-photorealistic virtual environments. *Proceedings of SIGGRAPH 2000* (July), 527–534.

KOWALSKI, M. A., MARKOSIAN, L., NORTHRUP, J. D., BOURDEV, L., BARZEL, R., HOLDEN, L. S., AND HUGHES, J. August 1999. Art-based rendering of fur, grass, and trees. *Proceedings of SIGGRAPH 99*, 433–438.

MAILLOT, J., YAHIA, H., AND VERROUST, A. August 1993. Interactive texture mapping. *Proceedings of SIGGRAPH 93*, 27–34.

NEYRET, F., AND CANI, M.-P. August 1999. Pattern-based texturing revisited. *Proceedings of SIGGRAPH 99*, 235–242.

OLIVEIRA, M., BISHOP, G., AND MCALLISTER, D. July 2000. Relief texture mapping. *Computer Graphics (Proceedings of SIGGRAPH 2000) 34*, 4, 359–368.

PERLIN, K., AND VELHO, L. August 1995. Live paint: Painting with procedural multiscale textures. *Proceedings of SIGGRAPH 95*, 153–160.

PULLI, K., COHEN, M., DUCHAMP, T., HOPPE, H., SHAPIRO, L., AND STUETZLE, W. June 1997. View-based rendering: Visualizing real objects from scanned range and color data. *Eurographics Rendering Workshop 1997*, 23–34.

RECKER, R. J., GEORGE, D. W., AND GREENBERG, D. P. 1990. Acceleration techniques for progressive refinement radiosity. *1990 Symposium on Interactive 3D Graphics 24*, 2 (March), 59–66.

ROCCHINI, C., CIGNONI, P., AND MONTANI, C. June 1999. Multiple textures stitching and blending on 3d objects. *Eurographics Rendering Workshop 1999*.

WILLIAMS, L. March 1990. 3d paint. *1990 Symposium on Interactive 3D Graphics 24*, 2, 225–233.

WINKENBACH, G., AND SALESIN, D. H. July 1994. Computer-generated pen-and-ink illustration. *Proceedings of SIGGRAPH 94*, 91–100.

WOOD, D. N., AZUMA, D. I., ALDINGER, K., CURLESS, B., DUCHAMP, T., SALESIN, D. H., AND STUETZLE, W. July 2000. Surface light fields for 3d photography. *Proceedings of SIGGRAPH 2000*, 287–296.

Figure 11: This model has two dependent views, one from the left and one from the right. The two dependent views fade in and out together based on the value of the blend function, specified on the left. Red points have a value of 1 (only the base-coat) white points have a value of 0 (use the dependent view).