EUROGRAPHICS 2000 / M. Gross and F.R.A. Hopgood
(Guest Editors)

Volume 19 (*2000*), *Number 3*

Real Time, Accurate, Multi-Featured Rendering of Bump Mapped Surfaces

M. Tarini[†] , P. Cignoni[‡], C. Rocchini[§] and R. Scopigno[¶]

Istituto Elaborazione dell'Informazione[||] ITALY, C.N.R., Pisa, Italy

Abstract

We present a new technique to render in real time objects which have part of their high frequency geometric detail encoded in bump maps. It is based on the quantization of normal-maps, and achieves excellent result both in rendering time and rendering quality, with respect to other alternative methods. The method proposed also allows to add many interesting visual effects, even for object with large bumb maps, including non-s rendering, chrome effects, shading under multiple lights, rendering of different materials within a single object, specular reflections and others. Moreover, the implementation of the method is not complex and can be eased by software reuse.

1. Introduction

Texture mapping has been widely used to enhance realism of computer generated images. The possibility to remap a two-dimensional color map over the projection of a 3-D polygon is crucial in order to produce rich color details in interactive time. Even low cost graphic hardware is now able to handle textures, and mapping is nowadays very efficient and, more importantly, the computation can be off-loaded from the main CPU.

Standard textures (i.e. rgb maps) add pictorial detail to the surface (like paint, labels and so on). Bump mapping techniques [1] have been proposed to add 3D detail (or at least a convincing illusion of it) to a surface.

It is useful to introduce now the terminology used in this paper: to be clearer in the illustration of the proposed technique, we will use the term **bump map** in a generalized way, meaning by it any 2D texture aimed to add 3D detail over a surface. Bump maps can be classified in three main different modalities:

- *displacement map*: each texel (texture pixel) stores a signed distance along the normal between the corresponding point on the polygon and the real surface (this is what is usually called bump map);
- *normal map*: each texel encodes the 3D normal of the point on the real surface corresponding to that texel;
- *light map*: each texel is directly the precomputed shade (RGB value, or just intensity value to be applied over a base color) for the corresponding point of the surface; a light map is just a static color texture used to approximate unmodeled geometrical details under a fixed lighting configuration.

Bump maps can be very useful to render high-frequency geometrical detail on a given object. Many objects can be represented by iterating a very small bump map over the surface (e.g. an orange skin or a bricked wall). Another, different use of bump maps is to permit the use of very low-complexity models that still appear rich of non-repetitive detail (as in Figure 1). For this purpose, the bump-map must be much larger and in general a bump texel is used only to represent a single point of the surface (our technique focuses on this use of bump maps). This type of bump map is obtained as a natural output in a number of modeling techniques: during mesh simplification, bump maps are useful for storing, in the simplified model, the geometrical detail present in the original model [6, 12]; in physical object acquisition, a normal map can be obtained by processing multiple

[†] Email: mtarini@di.unipi.it
[‡] Email: cignoni@iei.pi.cnr.it
[§] Email: rocchini@iei.pi.cnr.it
[¶] Email: roberto.scopigno@cnuce.cnr.it
[||] v. V. Alfieri 1, 56010 S. Giuliano T. (Pisa

photos (shot under different lighting conditions) of the acquired object [5, 18, 17]; displacement maps are computed from pairs of stereoscopic images [7].

This paper introduces a new simple and efficient technique to render bump maps stored in the form of normal maps. In short, at preprocessing time, the normal map is quantized; then, during rendering, a *light map* is computed on the fly starting from the quantized normal map through a lookup table.

In comparison with other approaches, our approach presents many advantages: computational efficiency, low memory cost, accuracy in the implementation of Lambertian reflection, possibility of adopting non-Lambertian lighting models, use of different lighting models within different parts of the same object. Also, rendering time does not depend much on texture size, allowing the use of large bump maps. Another advantage of this technique is that its efficient implementation is much eased by reusage: the preprocessing phase can reuse directly results and standard software intended for image processing (RGB palette quantization); the rendering phase uses common features of 3D accelerated hardware, that have been developed for different purposes - mainly to save texture memory (paletted texture). A first comparison with alternative solutions is presented in Section 2, and a more detailed evaluation is in the concluding section.

The paper is organized as follows. A brief overview of the state of the art is in Section 2. The proposed technique is outlined in Section 3. The two following Sections 4 and 5 show in more detail the preprocessing and rendering steps, respectively. Since most common approaches to render (and synthesize) bump maps are based on displacement maps, Subsection 4.1 describes, for completeness, a method to convert any displacement map into a normal map. Section 6 describes how multiple lighting models can be managed. In Section 7 we analyze the issue of adding color: like other approaches that use (computed-on-the-fly) *light maps*, we have to blend the shade information with the possible underlying color (typically a color texture). Finally, Sections 8 and 9 reports results and conclusions.

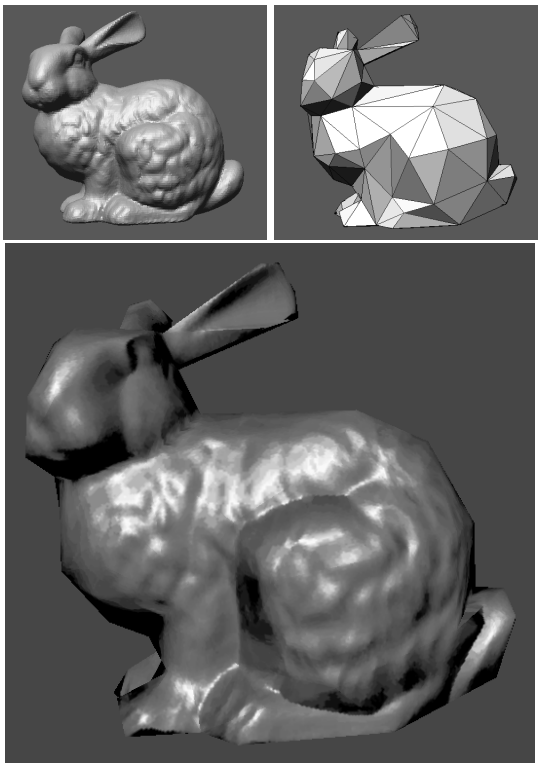

Figure 1: *The Stanford bunny. Top-left: a rendering of the original model (69.4 K faces). Top-right: a highly simplified model (251 faces), where most of the high frequency shape detail is lost. Below: real-time bump-mapped rendering of the same simplified mesh, produced using a table of 2048 normals.*

2. Related works

There are many ways to visualize a mesh whose polygons are enriched by some kind of bump-map. The methods can be divided in those that are intended for real time interactive rendering (like ours), and those that are intended for off line rendering.
Another distinction is on rendering results. There are two distinct effects that can be rendered to visualize the shape detail encoded in a bump map:

1. *shading* the surface of the object according to the encoded shape detail;
2. applying a *distortion* on the shape of the surface of the object, according to the encoded shape detail.

Let us call *shade based* those techniques that follow the first approach (and our approach is an example of such techniques) and *shape based* the techniques that follow the second approach or more precisely *shape-only based* those that only performs a shape distortion but not consider shading.

In the case of *shade-based* techniques, only the color intensity of the object will change according to the bump map. The approximation of the geometrical detail will be good if the geometry of the mesh is sufficiently accurate, and the best use of the bump-map is to store high frequency detail (like smoothness, roughness, bumps, small holes, cesel marks, discontinuities and so on).
Conversely, *shape-based* techniques modify also the silhouette, the shape and the parallax of the rendered object accordingly to the bump map. This can be useful for example for rendering impostors.
Note that any *shape based* techniques require displacement

maps, while *shade based* techniques in general can use any specific kind of *bump map*.

A very straightforward *shape-based* approach, in the case of displacement-maps, is to get rid of them by converting them into naked geometry. In fact, one can consider each texel as a vertex of a refined new geometry, and the new topology is simply derived. If a model has also a color texture, that it can be easily converted in the form of per-vertex colors. Following this approach, lighting and shading are simply postponed to the rendering phase and the visual results are very good. Obviously, this brute force technique has an huge overhead in space and rendering time and it is therefore suitable only for off line rendering.
A less brutal technique for *shape-only based* displacement map rendering has been presented in [14]. The main idea is as follows: the displacement map, during rendering time, is first converted into an ordinary texture, via a two pass 1-D transform. Then, the resulting bump-map is warped around the object with standard texture mapping. The basic disadvantage of this technique is the long time the first rendering phase takes with current hardware (even if authors claim that specific hardware could be designed to deal with it). Note that since no real time shade calculation is done, this technique is orthogonal, rather than alternative, to the one presented. In fact, instead of having a displacement map together with a color texture, one can imagine to have a displacement map, to be used as in [14], *and* the quantized normal map as in the technique proposed in this paper.

Shade based techniques are a very good alternative to *shape based* ones: they proved to give a very convincing illusion of bump while not being nearly as time consuming.
Shade based displacement-map rendering can be implemented by perturbing the normal of each pixel during rendering [1]; this approach relies on per pixel computations that, even when optimized, are very expensive. To overcame this problem, a number of approximations are introduced, most notably the tangent vectors are held orthogonal. Approximations of this per pixel process have been proposed, designed for hardware implementation. The products $p_u \cdot n$ and $p_v \cdot n$ between the tangent vector p_u and p_v and the surface normal n, for example, is assumed constant for each polygon in [8]. Similarly, p_u and p_v are supposed orthogonal and their length equal, and they are supposed to change slowly over the polygon in [15]. A common side-assumption is that displacement values are small. All this leads to artifacts, and therefore to a less convincing illusion of physical bump being present. Also, the lightning effects obtainable on a bump-mapped surface are limited to Phong and Lambertian shading.

Since hardware implementations are, *ipso facto*, expensive, many approaches, like ours, try to reuse existing hardware rather than designing new one.
The most straightforward *shade based* procedure to visualize a bump-map is to convert it, off-line, in a static *light map*, to be used as a color texture during rendering. The pros and cons of this are obvious: rendering is very fast, there is no time limit on the calculation of the lighting model and the light map can be blended with the color texture once and for all during preprocessing. On the other hand, this approach is limited to static lighting. Moreover, because we cannot consider the viewpoint position during the creation of the light map, we cannot compute view-dependent effects, such as the specular reflections.
Many techniques have been developed to obtain dynamic lighting. A common technique is an extension of an embossing algorithm for height fields [19], and consists in applying the same displacement map twice, as a color texture, shifting each time the position of the texture coordinates. The shift is computed, for each vertex, so that blending the displacement map with itself leads to an approximation of its derivative, which in turn is an approximation of the diffuse reflection. Specular reflection is not possible, unless the hardware gives some possibility to perform quickly a power rising for each texel (in that case, phong reflection takes two additional passes). Only very basic lighting models are rendered, and the light map calculated is only an approximation; some visual artifacts are produced especially if the displacements in the map have a considerable magnitude, or if the tangent vectors vary quickly over a polygon.
A more sophisticated *shade-based* technique using displacement maps was proposed in [13]. Again, it uses the same approximations as in [1], with the same visual results. For each frame, before rendering, a normal-to-shade function is regularly sampled over a unitary sphere, using polar coordinates, with a mathematical technique capable of quickly computing lambertian and specular reflection over a sphere for a distant light. During rendering the displacement map is transformed in a normal map (each normal expressed in polar coordinates): the result, cached for any subsequent frame, is used to obtain the shading taking for each texel the nearest value from the sampled sphere. Even considering these optimizations, computation is comparatively heavy, even if suitable for small texture maps. The precomputed normal-to-shade sampling used in [13] can resemble the index-to-shade table we use. Differences are in the sampling distribution of the table (uneven and independent from the shape of the rendered object, therefore requiring use of many more normals); in the synthesizing process of the table (which, being bigger, requires optimizations that limits the scope to lambertian and specular reflection and force the use of polar coordinates); in the way the table is used to transform the texture (not using any hardware and requiring to reload at each frame the texture on the graphic board from the system RAM).
A recent software technique for *shade based* bump-mapping is based on normal maps [9]. It can emulate only the Lambertian reflection for distant lights, but the emulation is totally faithful. The technique consists in a multiple pass rendering to obtain the light map (one pass for ambient and six passes for Lambertian). The normal map is divided into six textures, so that each one of its components (x, y and z) is stored in two

different texture (one for the positive and one for the negative values). The Lambertian reflection (that is, the dot product between the normal and the light vector) is obtained using the rendering transparency support of the graphic hardware: each pair of passes implements, directly on the display memory and all over the object, one of the three sums composing the dot product: $l \cdot n = (l_x * n_x) + (l_y * n_y) + (l_z * n_z)$. The ambient factor is computed with an extra, un-shaded pass. As usual, an additional pass (and an additional texture) is anyway required to add color. But the need to perform six rendering passes for Lambertian shading alone, plus one for each rgb texture, represent a severe overhead in time and space.

To finish, the work presented here has something in common to the technique presented in [2] and more recently in [10] where, in the final image, for each pixel some data is stored so that the rendered image can be quickly re-processed to simulate changes of the light position. For example, an index to a normal is stored in [2]); info about the reflected environment (light and a spherical polar representation of the reflected ray) is stored in [10]. But in both cases only a two-dimensional static view of the model can be dynamically lighted.

3. The proposed technique: an outline

Our bump-mapping rendering technique consists of a preprocessing phase and a rendering phase.
The goal of the preprocessing phase (see Section 4) is to obtain a quantized normal-map. Any normal-map can be quantized into one that uses only s_N different normals: at the end of this process we have a texture image where each texel is an index to one of the s_N entries on a lookup table N of normals. If the shape detail is mapped on the object via a displacement-map, than we first convert it into a normal-map (see Section 4.1).
In the rendering phase (see Section 5), for each frame, we first apply our lighting model (which starts from the current light direction and viewpoint position) on all the s_N normal vectors contained in the look-up table, obtaining the color table C which maps normal-indices to shaded colors. Then, using standard texture rendering features, we apply it automatically to each texel of the the indexed normal texture, transforming it in the proper light-map for that frame.
While all the normal maps used in the paper have been produced using the technique proposed in [4, 6], the technique proposed is very general and can be applied on any type of bump-map. An example of one of the normal maps used is in Figure 2.

4. Preprocessing: creating a quantized normal-map

As introduced above, given a normal map (encoded in object coordinate system) we want to quantize it, that is, to replace each normal by (the index of) one other normal chosen in a small set N of representative ones of size s_N . This, in general, introduces an error, since each normal in the normal map will be replaced by the "nearest" normal in N.
The choice of the parameter s_N is crucial: if we use only a few representative normals we obtain very fast rendering and preprocessing times, a low memory cost, but a worse normal approximation and image degradation. Conversely, the process becomes slightly slower but more high quality if a denser normal sampling is used. Empirically, for standard all round objects (the worst case), a value of 256 should be considered the minimum sufficient to produce sufficient results, when dithering is on and for solitary objects (see Figure 3). Excellent results are produced using $s_N = 2K$ normals, while for $s_N = 16K$ the approximation is in practice invisible.

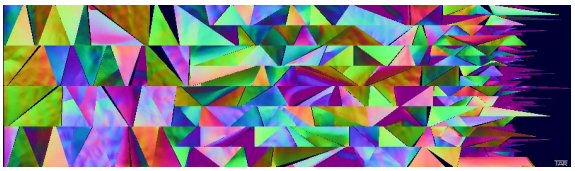

Figure 2: *Rather than showing the object (bunny mesh), we show here the corresponding texture map (1024x256 pixels, with normals mapped in the rgb space).*

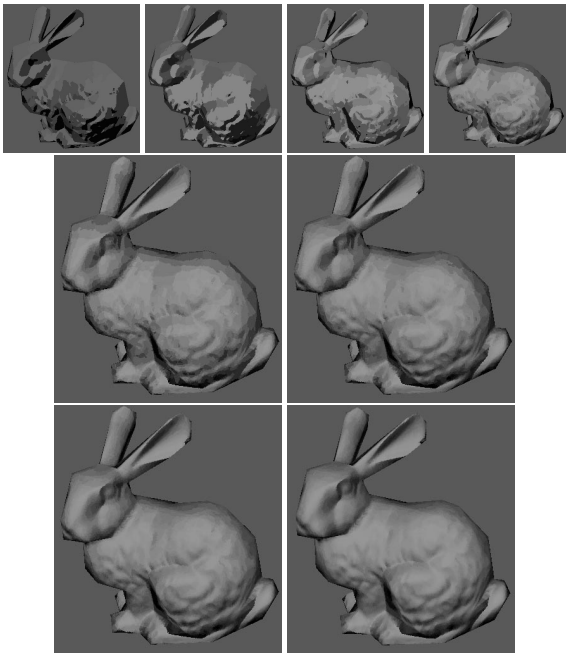

Figure 3: *Quality loss due to normal quantization. The same geometry (256 faces) is used in all the images; quantized normal maps with 16, 32, 64, 128 different normals are used above, 256 and 512 for the two images in the middle and 1024 and 2048 in the last two ones.*

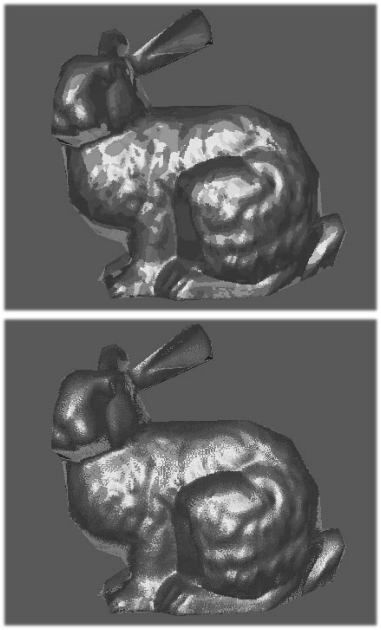

Figure 4: *Both images show the bunny bump-mapped using just 256 normals, but the image below uses a dithered normal map.*

The selection of N takes in account only the initial set of normals actually used in the object. This means, for example, that large flat surfaces will only have a sample in N, while complex bumped surface will have many more. In the example in Figure 1, for example, N will have a very few normal pointing downward, but many more pointing somewhere upward (the lower part of the bunny is almost flat).
Normal quantization can be easily obtained just reusing standard image processing software. In fact, a normal composed of $x\ y\ z$ components, can be stored directly as an $r\ g\ b$ pixel (mapping the interval $[-1..1]$ into $[0..255]$). Once the normal map is stored as a 24 bit image, it is then sufficient to transform the obtained image into a paletted image (setting the palette size to be s_N). The color palette computed in the process represents the set N of normals. Obviously, we must remap back each entry of the palette from $[0..255]$ to $[-1..1]$ and then renormalize it, but this is again a fast process due to the small size of N.

As noticed in many papers on color quantization, dithering is very useful to reduce the visual impact of the approximation, (especially if we use a small palette). Applying dithering to normals produces as well an improved image quality (see Figure 4).

4.1. Displacement map to Normal map

This section shows, for completeness, a simple way to transform a *displacement map* into the corresponding *normal map* required by the proposed technique. This is a preprocessing phase, that has to be performed only once for each map. Time efficiency is thus not critical; therefore, we can spend some time to obtain a precise conversion, which is important with respect to visual quality.
A good approach to do this conversion is to find, for each texel t mapped on the surface, the plane fitting the points corresponding to t and to other q texels displaced in the neighborhoods of t (e.g. for $q = 4$, we take the texels on the right, on the left, above and below.).
Working on a coordinate system with the x and y axis on the surface of the polygon, and z along the normal, the problem becomes a two-dimensional one: the plane can be parameterized with a triplet (a_0, a_1, a_2) (for (x, y, z) on the plane, $a \cdot (x, y, 1) = z$). The best fitting plane is found by applying LMS to minimize $(Xa - d)^2$, where X is the matrix composed by q rows, one for each texel i, of the form $(x_i, y_i, 1)$, and d_i is the (scaled) displacement value at coordinate x_i and y_i. The obtained vector $(-a_0, -a_1, 1)$ is then translated on the real word system, normalized and stored in the *normal map* at the corresponding texel.

5. Rendering quantized normals to color

To transform quantized normals into dynamically shaded colors, we need to introduce a small modification to the standard rendering pipeline, by introducing a new view-dependent phase. Our enhanced renderer, when loading the indexed normal map associated to an object in the scene, retrieves the normal table N from the palette of the image on which the normal map was stored. For each frame, rendering is subdivided into two phases:

1. the normal lookup table N is transformed in a shaded color lookup table C by applying to each entry of N the current lighting model; since the normals in N are encoded in object coordinate space, the vectors considered in this software computation of the illumination model (e.g., current light direction and view direction) are first transformed using the inverse of the current modeling transformation;
2. the surface of the object is rendered, using the classical graphic pipeline; the lighted and shaded surface is then computed by mapping the indexed normal texture, which now indexes a shaded table C, over the mesh. This mapping phase is standard and does not introduce overheads. Many hardware architectures, in fact, give hardware support for paletted textures, and an explicit step is provided in the raster pipeline to translate indexes into RGBA colors.
   If a given graphic board lacks this particular support, performances may slow down considerably. In this case only, we are forced to perform the index-to-RGBA mapping "by hand". The software renderer has in this case an added burden: for each frame, it has also to convert on the fly the indexed texture into a standard rgb

texture; then the renderer reloads the new shaded texture from RAM into texture memory, to produce the next frame (actually, OpenGL provides support to apply the mapping C directly while reloading the indexed texture). The overhead in this case is mainly the time necessary to download the new texture from RAM to the graphics board texture memory. On slower machines, if the frame rate drops too low, it is always possible to give up real time lighting by stopping doing the conversion from indexed normals to colors (or by using each light map for some consecutive frames).

The first step takes a very short time, and is performed by the CPU, while most of the rendering time is taken by the second one, performed by the graphic hardware (see times reported in Section 8).
If multiple independent bump-mapped object are contained in the same scene, the first step must be repeated for each each of them.

5.1. Computation of color tables

The color table C is produced by applying to each normal n in the set N a function $f(n) \rightarrow c$. The function can implicitly take in account other information, typically the light vector (or a set of light vectors), the light color, the view direction for the given frame, and so on. Some of those vectors (the light vector, the view direction) need to be transformed, before computing the table, in the object coordinate system. However, all these variables are forced to be constant, for a given frame, all over the model, and for some of them this represents an approximation [11].
Depending on the modality to assign colors adopted (see Section 7), the function $f(n)$ can return: an RGB value, to be applied directly over the object; an intensity value, to be blended with the underlying color; an intensity value together with the RGB values of the specular reflected light.
The function f actually used corresponds to the lighting model chosen. There is a particularly wide gamut of possibilities, since f is computed only s_N times (a relatively small number) and therefore the time spent computing C is anyway short. Just to cite some examples:

1. ambient factor and Lambertian reflection, using one or more lights (as in Figure 3);
2. specular reflection component (as in Figure 1), again with any number of light;
3. environment mapping (as in Figure 5*), where entries in C are taken directly from an environment map. If the environment map is very detailed, to produce high quality results we may be induced to adopt a normal table N bigger than usual;
4. cartoon-like rendering, where we adopt an f function that maps normals in a very restricted range of colors. This is different than setting a restricted number of normals in vector quantization, because in the first case not only the shade but also the the shape of uniformly colored zones changes as light moves;
5. non realistic rendering using any non physical but interesting function from an information visualization point of view (like the variant of the reflection function in Figure 6);
6. chrome effect (as in Figure 7);
7. any combination of the above.

Figure 5: ** An example of environment map is shown; the environment mapped on the vase mesh is the one contained in the image above (a photo shot to a Christmas tree ball, with one of the authors specularly reflected).*

5.2. Mipmapping

If the mip-mapping technique must be applied, then the various levels of Mip-mapping for the quantized normal-map cannot be computed by the renderer as usual, since that texture consists of indexes, that cannot be directly averaged. For mip-mapping purposes, the "average" of 4 adjacent texels $t_1, \ldots t_4$ (which are indexes to normals in N) must be defined as the index in N that best approximates the average of the 4 normal $N(t_1), \ldots N(t_4)$.

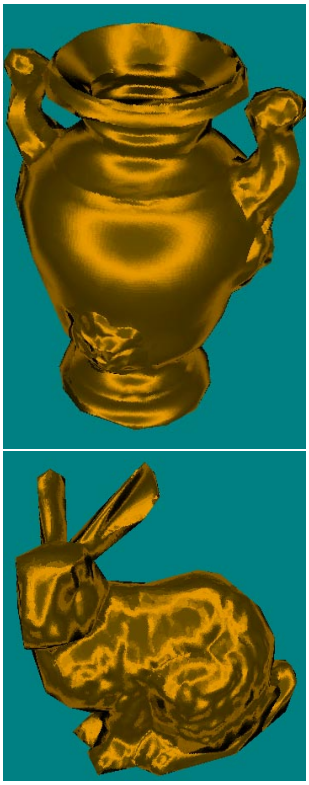

Figure 6: *Examples of bump-mapped objects rendered under a non-realistic specular reflection. In this example the reflection is computed by doubling the angle between the reflected ray and the view direction in the Phong reflection formula.*

Figure 7: *Examples of bump-mapped objects rendered using a chrome effects. In this example the color is assigned by taking into account the reflection of a typical landscape (brown below, and blue above).*

6. Multiple lighting models

It can be useful to support the use of multiple lighting models on different parts of the same object. This allows, for example, a metallic part and a plastic part to coexist on the same geometric object. The number m of lighting models used should not generally be too high; 2 or 3 is usually enough, but values of 8 or more are sometime useful and can be dealt with.

To obtain multiple lighting models each object should contain information about all the m lighting models used, including values for ambient factor, Phong coefficient, specular reflection color, diffuse reflection color (if color is not specified in a color texture), and so on. Each normal in the texture should include an extra field for the *material*, specifying which lighting model has to be applied to that normal when the color table C is evaluated: the same normal now can in principle appear in the table even up to m times, with different values for the material.
Actually, the *material* field is not stored explicitly in the normal table N. We designed an approach that sorts the pairs (material,normal), once in the preprocessing phase, in order of *material* and then throws away the *material* field keeping record only of the intervals. This is not only to reduce the space overhead, but also to avoid time overhead (an *if-then-else* like statement) in the computation of each entry of the color table C.
Once obtained the table C, the rest of the rendering algorithm remains the same. Note that managing different materials does not introduce an overhead *per se*; a space and time overhead can be introduced only when a larger set of normals N become necessary, to give enough precision to the representation of a set of (material,normal) pairs, which is in general wider than the pure set of normal.

To further reduce the overheads for adding materials, it is convenient that the preprocessing normalization of the normal texture takes in account the material as a forth, discrete component of the normal vector. For example, look at the metal and plastic bunny on the top of Figure 8: most of the normals pointing up are *"metal"*, while most of those pointing down are *"not metal"*.
In the color quantization phase, the *material* attribute can be stored in the α component of the *rgb*α pixels of the image representing the normal map. Alternatively, if the color

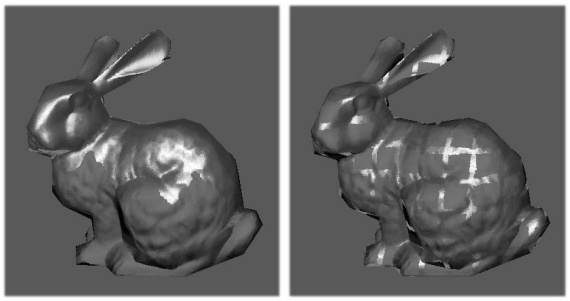

Figure 8: *An example of a model containing a metal and a plastic section (in this case, specular reflection is the only perceptive difference).*

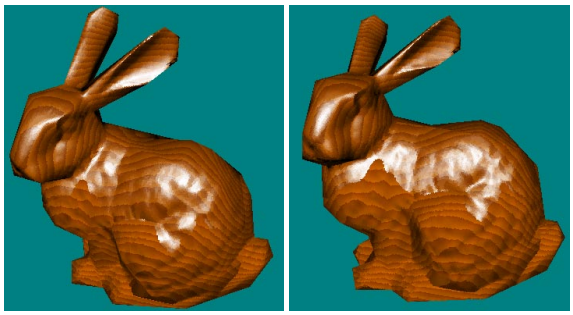

Figure 9: *A partially painted wooden bunny is shown. The top half of the bunny is painted with a transparent shiny paint (high specular reflection), while the bottom half is glossy. The wood grain is stored in a color texture, which is blended with the shade obtained from the normal map. The normal map uses 2048 normals subdivided in two materials, and blending is done is a single second pass.*

quantization software available lacks the support for the alpha component, one can use a different mapping from the n_x n_y n_z components of each texel of the normal-map into r g b values of the pixel of the image: the color cube can be subdivided in 8 equal sub-cubes, of which 4 are reciprocally not adjacent, so that each normal space of each material (up to four) can be mapped in a different, non-adjacent color space sub-volume.

7. Blending color

The object to be rendered may have defined some information on the pictorial detail, or color by short. This detail must be integrated with the results of view-depedent shading. In this section we assume that only the simple lighting model composed of specular and diffuse reflection is evaluated on the object.

There are at least three sub-techniques that can be adopted, depending on the need of the current application and the nature of the object rendered. They take respectively three, two and one rendering passes to integrate the base color and the results of the lighting model.

7.1. Exact specular reflection

For each n_i in the normal table, we can process it by applying the lighting model and storing the result in the color table C as follows. The lambertian shade intensity can be stored in the α_i component of the entry c_i, while the reflected light color (already multiplied for the specular factor raised to the required power) are stored in the r_i, g_i and b_i components. For each object surface parcel having quantized normal n_i, the final color quadruple ($rgb\alpha$) can be computed at rendering time as:

$$\begin{array}{lcl} r & = & r_o \cdot \alpha_i + (1 - r_o \cdot \alpha_i) \cdot r_i \\ g & = & g_o \cdot \alpha_i + (1 - g_o \cdot \alpha_i) \cdot g_i \\ b & = & b_o \cdot \alpha_i + (1 - b_o \cdot \alpha_i) \cdot b_i \\ \alpha & = & \alpha_o \end{array} \qquad (1)$$

where $(r_o, g_o, b_o, \alpha_o)$ is the base color of the object (which, for example, can be encoded in a standard $rgb\alpha$ texture).

Unfortunately, it is well known that the current standard OpenGL implementations do not provide blending functions that are complex enough to compute the formula above in a single extra pass. Two extra passes are required: one to multiply the original color with α_i and another one to add the second term of the sum. Needless to say, the last one can be avoided if no specular reflection is needed.

7.2. Approximate specular reflection

If an approximate specular component is enough, the blending formula (1) can be replaced by the simpler

$$\begin{array}{lcl} r & = & r_o \cdot \alpha_i + (1 - r_o) \cdot r_i \\ \ldots & & \end{array} \qquad (2)$$

which can be computed in a single extra pass (see an example in Figure 9).

The visual result gives an acceptable approximation, especially when the light position and the view position are not too far apart one to the other.

7.3. Encoding color as a material attribute

If the object is subdivided into a small number of areas with uniform base color (like for example most CAD models), then each color can be codified in a different *material* and both color and shading can be rendered in the same pass (see Figure 10* for an example). In this case, the final color (lambertian added to the reflected component) is directly stored in the $r_i g_i b_i \alpha_i$ components of the color entry c_i, and no blending is needed at rendering time.

As a positive side effect, each color in the model can be also associated with a different material attribute (e.g. the specular reflection factor), or even with different lighting effects (chrome, environment mapping, etc).

Figure 10: * *Model of a (faked) greek vase. The naked geometry is visible in the first column (flat shaded above and Gouraud shaded below). The images in the second column show some examples of single-pass rendered images of the vase enhanced by a quantized normal map that uses an 8K table and encodes both normal and color (note the different specular component associated to each of the two color). In the last column two different normal maps are used to add different 3D details: in the image above the paint is made "thicker", below some engravings are also added (and the depressed parts are made much rougher than the rest).*

8. Results

The models used within this paper have been obtained by simplifying some detailed original models with the Jade2 simplification tool [3]. The corresponding normal-maps have been obtained with *PASTex* [6], a software which retrieves the detail lost during simplification (either shape or color) and encodes it in textures (bump maps of any kind) mapped on the surface of the simplified object. To support the need of this particular experimentation, the original *PASTex* tool has been extended to support the construction of normal maps with encoded materials (or to introduce added detail in the texture construction process, like the greek-like painting mapped on the vase in Figure 10* and the vase engraving).
Normal-map quantization has been done using *PPMquant* [16], which is part of *NetPbm*, a freeware, multi-platform toolkit for image format conversion and basic image processing.
Finally, rendering of bump mapped object has been performed using *PlyView*, a MS Win9x OpenGL renderer that has been developed by the authors.

The results of an empirical evaluation of our approach are reported in Table 1. The results reported in the table have been obtained by using the *PlyView* renderer running on a Pentium 300 MHz with an NVIDIA Riva 128 card. The table shows, depending on the number of normals used, the cost of the per-frame overhead (to compute for each frame the new color palette C, measured in milliseconds) and the total frame rate (fps) obtained on the architecture used to render the 251-faced bunny model. These values have been evaluated for the different lighting models and effects used.

	Lambertian		Lambertian & Specular	
Number of normals used	per-frame overh. (msec)	fps	per-frame overh. (msec)	fps
256	0.0	16	0.1	16
2048	0.3	16	1.2	15.5
16348	3.9	13	12.5	11

	Chrome		Environment Mapping	
Number of normals used	per-frame overh. (msec)	fps	per-frame overh. (msec)	fps
256	0.0	16	0.1	16
2048	0.7	16	1.7	15.5
16348	8.6	12	18.3	9.5

Table 1: *Empirical evaluation of the proposed approach, produced on a Pentium 300 MHz with an NVIDIA Riva 128 board.*

Note that because of cache coherence the time needed is not linear with the number of normals.
To give a rough evaluation of the amount of extra rendering time consumed due to the use of a *paletted* texture, consider that the same object rendered under a static lighting (i.e. using a precomputed shade-map, that is mapped as a common color texture) is visualized at 16 fps. Another useful comparison is against the original bunny model (visible in the top-left part of Figure 1), which is a pure geometric mesh (69K faces) with no textured detail, is rendered at less than 2 fps.

9. Conclusions

We have presented a technique to perform real time bump-mapping in an efficient and accurate manner. It is simple in the concept, and easy in the implementation, but despite that it leads to very good results, allowing a low-cost real-time rendering of shape detail of an object surface. In comparison to other real-time bump-mapping techniques, it requires less rendering passes and results in more accuracy. Moreover, it provides a notable flexibility and can be easily extended to achieve easily a wide gamut of interesting possible rendering effects, including non-standard lighting and shading effects, rendering of both color and shading in a single pass, shading under multiple lights, rendering of multiple materials within an object (each with its own properties, and disposed in arbitrary patterns), specular reflections and so on.

The technique is based on quantization. Quantizing the normals of an object means, in a sense, to force the quasi-coplanar parts of it to be merged into larger irregular shaped flat parts. Therefore, the technique presented can be seen as a simplification process applied over the object, similar to the classic geometry-oriented simplification techniques, but with some important differences:

1. it is done over the texture, not the geometry;
2. resulting artifacts consist in both cases in the perception of iso-valued sections, but in our case those look much more natural, because they are in general irregularly shaped;
3. the ratio of the reduction in complexity over the loss in quality is much better; this is also because even if we use a reduced number of different normals, their distribution on the surface can produce many different patterns and enhance considerably the shape detail perceived.

Moreover, normal quantization can be used in addition to surface simplification: given a very detailed mesh, it is convenient to simplify it, then reproduce the detail lost in a normal texture, and finally quantize the resulting normal texture. The final textured mesh is much less expensive to be rendered (and to be stored) than the original one, while the appearance remains very similar (see Picture 1).

Obviously, normal quantization reduces considerably rendering costs. A simple, software-based Phong renderer evaluates the lighting model (that is, a normal-to-color computation) for each pixel covered in the rendered image. In the case of a standard hi-resolution image (say $1.000 \cdot 1.000$ pixel) covered by the model with a 50% fill rate, roughly we have thus 500K evaluations. The same applies also to all bump map renderer that uses per-pixel computations. Similarly, if we adopt an approach where the texture is shaded, than the complexity is linear to the current texture size. As an example, the texture associated to the bunny model is around 250K texel wide, and therefore a renderer that uses per-texel lighting computation would require 250K evaluations. With the presented technique, instead, very good results are obtained with at most a few thousands normal-to-color computations.

The advantages of the proposed technique, in particular if compared to similar hardware-oriented solutions, can be summarized as follows:

1. *rendering speed*: the per-frame preprocessing requires few milliseconds, and HW-assisted rendering requires only one or two extra passes (apart for the base color one), depending on the technique used. Not only this is much less than many bump-mapping techniques, but it is also possible with the a single pass to merge pictorial and bump detail;
2. *flexibility (different lighting models)*: it is possible to use different kinds of lighting models, such as: simple Lambertian reflection, specular reflection, or non standard lighting or shading models, chrome, etc.;
3. *flexibility (multiple lighting models)*: it is possible to apply different lighting models over different parts of the *same* object;
4. *low space overhead*: a quantized normal-map takes only

two bytes per texel (or even one, depending on the quantization factor), not to mention that we can include also the color information in the table (in the case of a discrete color set);
5. *rendering cost distributed between the CPU and the graphic hardware*: during rendering, only the first [much faster] phase, is performed by the CPU while the standard graphics pipeline computation are performed by the graphic subsystem at HW-assisted speeds;
6. *easy of implementation*: this is due to software reuse, in particular color quantization.

The method presented has some limitations, that can be summarized as follows.
A specific hardware features is required to obtain efficient rendering (i.e. paletted texture support). Note that this is a standard feature in most graphic hardware, but it is typically used to - and was originally intended for - save texture ram space, which is a less and less precious resource. Therefore, some recent graphic hardware systems do not support this feature. This could prove a short sighted policy, since paletted textures can be very useful, not only in the way described in this article, but also in many other real time procedural texture mapping techniques.
Like many other similar techniques, also the presented one has some intrinsic limitations or introduces some aliasing:

- both light position and view position are considered to be constant all over the object. While this disadvantage cannot be overcome, the visual impact is usually very low;
- the quantization error may introduce some aliasing, which can be reduced beyond visibility (if the related overhead in space and time is considered acceptable) by increasing the number of normals;
- specular reflection is approximated in the single pass approach, but can be evaluated correctly with an extra rendering pass, **or** by storing color as material, **or** by avoiding either specular reflection or color information.

Acknowledgements
We acknowledge the financial support of the Progetto Finalizzato "Beni Culturali" of the Italian National Research Council. The *bunny* dataset is courtesy of the Computer Graphics Group at the Univ. of California at Stanford.

References

1. J. F. Blinn, *Simulation of wrinkled surfaces*, Computer Graphics (SIGGRAPH '78 Proceedings) **12** (1978), no. 3, 286–292.
2. E. Catmull and A.R. Smith, *3-D transformations of images in scanline order*, Computer Graphics (SIGGRAPH '80 Proceedings) **14** (1980), no. 3, 279–285.
3. A. Ciampalini, P. Cignoni, C. Montani, and R. Scopigno, *Multiresolution decimation based on global error*, The Visual Computer **13** (1997), no. 5, 228–246.
4. P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno, *A general method for recovering attribute values on simplifed meshes*, IEEE Visualization '98, IEEE Press, 1998, pp. 59–66.
5. _____, *Pictorial detail acquisition and patching on 3d objects*, Rendering Techniques '99 (G.W. Larson ED. Lischinski, ed.), Springer KG, Wien New York, 1999, pp. 119–130.
6. P. Cignoni, C. Montani, C. Rocchini, R. Scopigno, and M. Tarini, *Preserving attribute values on simplified meshes by re-sampling detail textures*, The Visual Computer **15** (1999), (to appear).
7. P.E. Debevec, C.J. Taylor, and J. Malik, *Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach*, SIGGRAPH 96 Conference Proceedings (Holly Rushmeier, ed.), Annual Conference Series, ACM SIGGRAPH, Addison Wesley, August 1996, pp. 11–20.
8. I. Ernst, D. Jackèl, H. Rüsseler, and O. Wittig, *Hardware-supported bump mapping*, Computers and Graphics **20** (1996), no. 4, 515–521.
9. C. Everitt, *Orthogonal illumination maps*, Paper written for OpenGL.org. Available on the web at: `http://www.opengl.org`, 1999.
10. P. A. Fletcher and P. K. Robertson, *Interactive shading for surface and volume visualization on graphics workstations*, Proceedings of the Visualization '93 Conference (San Jose, CA) (Gregory M. Nielson and Dan Bergeron, eds.), IEEE Computer Society Press, October 1993, pp. 291–299.
11. J. Foley, A. van Dam, S. Feiner, J. Hugues, and R. Phillips, *Introduction to computer graphics*, Addison Wesley, 1993.
12. V. Krishnamurthy and M. Levoy, *Fitting smooth surfaces to dense polygon meshes*, Comp. Graph. Proc., Annual Conf. Series (Siggraph '96), ACM Press, ACM Press, 1996, pp. 313–324.
13. G. Miller, M. Halstead, and M. Clifton, *On-the-fly texture computation for real-time surface shading*, IEEE Computer Graphics & Applications **18** (1998), no. 2, 44–58.
14. M. Oliveira and G. Bishop, *Relief textures*, Tech. Report TR99-015, University of North Carolina, Department of Computer Science, Mar. 1999.
15. Mark Peercy, John Airey, and Brian Cabral, *Efficient bump mapping hardware*, SIGGRAPH 97 Conference Proceedings, Annual Conference Series, ACM SIGGRAPH, Addison Wesley, August 1997, ISBN 0-89791-896-7, pp. 303–306.
16. J. Poscanzer, *Ppmquant*, Part of

NetPbm, Avaible on the web at: `http://wuarchive.wustl.edu/graphics/packages/NetPBM`, 1991.

17. H. Rushmeier, F. Bernardini, J. Mittleman, and G. Taubin, *Acquiring input for rendering at appropriate levels of detail: digitizing a pietá*, Eurographics Rendering Workshop 1998 (G. Drettakis and N. Max, eds.), Springer Wien, June 1998.

18. H. Rushmeier, G. Taubin, and A. Gueziec, *Applying shape from lighting variation to bump map capture*, Eurographics Rendering Workshop 1997 (P. Slusallek J. Dorsey, ed.), Springer Wien, June 1997, pp. 35–44.

19. John Schlag, *Fast embossing effects on raster image data*, Graphics Gems IV (Paul Heckbert, ed.), Academic Press, Boston, 1994, pp. 433–437.