INSTITUTO NACIONAL DE MATEMÁTICA PURA E APLICADA

Editing RGBNs

Thiago Pereira
Luiz Velho

Rio de Janeiro, January 30, 2009

Abstract

A color image with an associated normal stored for each pixel is a useful datatype. It improves on images since geometry information is also present while at the same time it avoids all the complexity present in a 3D mesh. In this report, we investigate editing these enhanced images. Local shape deformation operators are presented. Global operators are developed using texture synthesis methods. We also propose a new filtering technique.

Keywords: image processing, geometric modelling, normal maps, filtering, texture synthesis.

Contents

List of Figures

Chapter 1

Introduction

The quest for realism has always been the grail of computer graphics. Generating photoreal images requires a great knowledge of light and its interactions with objects in the scene. It also requires very good descriptions of objects. Shapes in the real world are very detailed, as such it is crucial to have computer models which can represent well even the finest turns in a surface. In this work we investigate the editing of color and normals images.

1.1 Representing details

Rendering complex high resolution models is a very difficult task. When represented as polygonal meshes, these models include so many small triangles that it overloads the graphics rendering pipeline, both in terms of time and space efficiency. It is common to view the modeled object in a multiresolution framework that allows lowering or increasing the level of detail of the model. With such a continous multiscale representations, it is easy to obtain a very simple model for objects that are far from the camera for example. Such objects will contribute to few screen pixels and do not need to be rendered fully.

One problem with these multiscale schemes is that they treat all scale levels equally, all scales are represented as meshes in the above example. Different geometric scales contribute in different ways. Lighting provides a good example to

illustrate this concept. Coarse scale geometry (position) influences how light intensity dampens while it goes from the light source to a lit model. Middle scale (normals) are also geometric information, yet a surface point will be less lit if its normal is away from the light direction, its exact position is less relevant. Fine scale geometry (microfacets) will influences how incoming light and outgoing light are related in the illumination hemisphere. The equations that govern each geometry scale behaviour are different and so are their storage requirements. A multiresolution hierarchy which does not handle all scales the same way is thus necessary.

It is common to use three different levels [4]. The Macrostructure level is the coarsest one. It is usually represented as a triangle mesh or a spline surface. In this level is the general shape of the model as would even be seen from a distance. The finest level is Microstructure. Elements in this level are so small that they cannot be distinguished, such as microfacets. The Mesostructure level contains intermediate geometric details that are still visible with a naked eye such as bumps and creases.

It remains an open question how to represent geometry mesostructure. The most simple way is to use a triangle mesh. If the triangles are small enough, mesostructure will be represented. Since a mesh does not assumes regular sampling, it must store the connectivity of its vertices and not just the 3D position of each vertex. This makes more storage necessary. On the other hand, texture maps do assume a regular sampling pattern reducing storage requirements. Textures not only save space, but also time. The more structured the information is the better optimized algorithms can be, even to the point of exploiting memory coherency at the hardware level.

A texture map can store many different color attributes like diffuse and specularity colors, but also geometric attributes. Storing geometry in textures has some advantages over meshes. It is easy to leverage the power of multiresolution, as described above, with the mipmapping technique. Most GPUs support this feature. For very detailed objects a full mesh representation will include many vertices, usually more than the number of pixels in a rendered image. If there is information in a texture it will only be consulted on a per-pixel basis and the rendering pipeline will be faster.

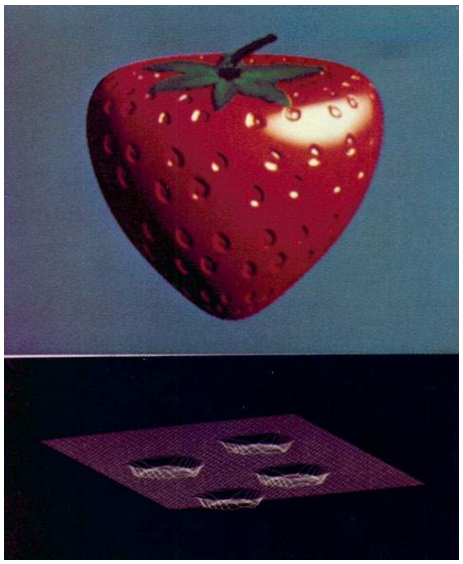

Figure 1.1: In Bump Mapping a deformation (below) is added to the straberry surface in the normal direction. Images taken from [1].

The arguments above show that storing a texture is better than a full mesh. One question remains: what geometric attribute should be stored? In what follows, we will refer to the normal calculated from the base geometry as the base normal. We are not interested in which way this normal was obtained. It could be a polygonal face's normal, an interpolation of vertices normals or anything else. In Bump Mapping [1], a height map is used. This means beyond the base geometry, texture stores a displacement in the normal direction (Figure 1.1) . The normal of this local height map is calculated and added to the base normal to obtain the final normal used for lighting purposes. Traditional bump mapping requires a well defined tangent plane where the height map lives and derivatives will be taken. Instead of taking derivatives, one can also store the final normal itself. One advantage is that no tangent plane needs to be stored. This technique is called Normal Mapping and is detailed in the next subsection.

Notice that in bump mapping (and normal mapping), only the normal is affected and not the geometry itself. While shading will look perfect, the model's silhouette is not affected [1]. In Displacement Mapping this is not a problem since the local

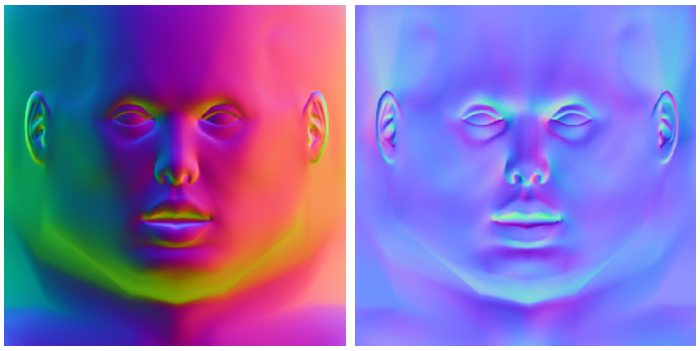

Figure 1.2: Object-space normal maps (left) have range in the unit sphere. Tangent-space maps (right) take values on a single hemisphere. Images taken from [2].

height map is used to change position itself.

1.2 Normal Maps

Normal mapping stores the normal in a texture map. During rendering, this map will be consulted and used to calculate lighting. Since this normals will be stored at a much higher resolution than the geometry itself, the model will look much more detailed. These quality results bring an efficieny penalty. Yet performace gains are obtained over bump mapping which requires some calculations to get to the final normal.

The normals can be represented in two different ways [2]. In object-space normal maps they are stored in an object coordinate system, assuming values in the entire unit sphere. Tangent-space maps store normals in a tangent space, as such its range is a single hemisphere. This tangent vector basis usually consists of interpolated vectors assigned at vertices.

Usually a 3D unit lenght vector represents the normal. While it is possible to use a floating-point texture, normal maps are usually signed or unsigned integer textures. The range of (x, y, z) coordinates in a unit normal are limited to $[-1, 1]$ and this interval is mapped to $[0, 255]$ and stored in an unsigned byte. The map can thus be opened is most image editors which will interpret each normal as a

color in RGB space. Since the vector $(0, 0, 1)$ is mapped to the color $(128, 128, 255)$, a predominantly blue color, most tangent space normal maps look blue. Since we used a uniform quantization of the unit cube to represent unit vectors, we are loosing much of the resolution provided by 24-bit textures. In [2] other representations are discussed.

Now that we know how to represent normal maps, we are left to answer how do we get our hands on them? The authors of [5] after simplifying a large mesh, store color and normals in textures in a way that preserves appearence (rendered image). This means pixels on screen should deviate little from the original color. After building a parametrization, the normals are obtained by rendering normals as colors into parameter space (texture). In [6], attributes are transfered from the high-resolution mesh, by sampling the faces of the low-resolution mesh and projecting them onto the high-res mesh. The projection step is performed with point-to-face distance calculations. The high-res mesh can be created with a modelling software, followed by mesh simplification producing the low-res mesh. This task is made easier by using subdivision surfaces.

Capturing real-world objects is an alternative. By capturing the high-res mesh with 3D scanning technology, the normals can be derived from position, but derivatives will enhance noise. Photometric stereo that allows for capturing normals directly will be discussed in the next section. Normal maps can also be built with procedural methods [7]. Texture synthesis is another approach which we discuss in Section 1.6.

In most applications of normal mapping, there will also be a color texture available. If we think of the attributes themselves and abstract the mapping process, we are left with images that contain color and normals. Editing these RGBN images are the focus of this report. Editing normal maps in a regular image editing software is difficult. First because of the distortions introduced by parametrizations which are hard to comprehend by the user. Second because normals are not colors.

To counter the first problem we assume a projective mapping from the surface into texture space, a picture of a camera for example. This mapping has the ad-

vantage that users are used to projective mappings and so editing is more intuitive. Projective mappings also help solve the second problem. As will be seen in the next sections, it makes many normal operations easier like filtering.

One problem with projective mappings is that it is not possible to continuosly map a closed 3D surface onto the plane. In future work, an atlas of projective normal maps will allow editing any model. By now we assume we are editing an RGBN image of a model taken from one fixed viewpoint.

1.3 RGBN

Toler-Franklin et al. coined the term RGBN to refer to an array of pixels with associated color and normal channels. We illustrate below some applications of RGBNs, but the power of this data type abstraction is being application independent.

1.3.1 Advantages of RGBNs

Obtaining quality 3D geometry (positions) of an object requires a 3D scanner, still a very expensive device. Even the best of those, still lacks resolution being far behind the much cheaper digital cameras. Noise is also a problem, leading to a trade off between low noise and higher resolution which further decreases the final quality. On the other hand, capturing only a model's normals is easier using shape from shading (SFS) techniques [8]. These methods take a shaded image as input and aim at inverting the illumination equation to obtain shape. Since SFS works on regular images, it allows very high resolution models to be obtained.

In some SFS methods, normals are first obtained from raw data and then undergo an integration process to obtain positions. The integration step is unstable. As Nehab et al [9] points out, reconstructing geometry from normals brings low frequency errors. For RGBNs we are only interested in the normals, as such we profit from SFS advantages without facing its drawbacks.

As detailed below, acquisition of normals is slightly more complex than usual photography, yet the power and flexibility of 3D models can be attained. We next

show some applications, that highlight the possibilities and limitations of working with RGBNs. As described above, recovering positions from normals can be unstable. Some works avoid using positions by working only with normals. In image relighting the normal map is used to fit a local lighting model [10] and rendering is done per-pixel. Many NPR rendering algorithms have been shown to work even if only a normal map is available [11]. These include toon shading, line drawing methods, curvature shading and exaggerated shading. RGBNs are thus a very good tool for understanding real models as they easy to capture and easy to analyse.

RGBNs are limited in that they do not allow for change of viewpoint or realistic shadow calculations. Both these limitations are a consequence of having only a local description of the model (normals) instead of a global one (positions). As described above, a normal mapped 3D mesh with a projective atlas can counter these problems. The lower resolution position information in the mesh vertices compensates these RGBNs issues, while the high resolution normal map builds on its advantages.

1.3.2 Acquisition Detail

Shape from shading methods are the most common technique for acquiring high-quality normal maps. In the simplest version of SFS, one looks for computing a 3D surface from a grayscale shaded image (photograph). The problem was first posed in [8] who formulated it as solving a non-linear PDE. We know today that SFS is not a well posed problem. By varying albedo or lighting conditions, different surfaces can yield the same shaded image. For these reasons, it is common to assume uniform albedo with Lambertian reflectance and a known light source.

Photometric Stereo (PS) improves on simple SFS. It does not assume constant albedo and to compensate for that, it requires multiple images from the same view point as input, each illuminated with different known light positions. For now, let's assume a single color channel. PS solves for the normal vector at each pixel $p = (x, y)$ separately using a Lambertian lighting model:

$$I_p = a_p < n_p, l_i >$$

where I_p is the intensity, a_p is the scalar albedo, n_p is the normal vector at pixel p. Also l_i is the direction of a light source assumed far away in image i.

We can rewrite this equation as:

$$I_p =< (a_p n_p), l_i >$$

There are 3 variables in the vector $a_p n_p$, as such 3 images would lead to 3x3 linear system having a single solution. Usually more than three images (and light sources) will be used making the method more robust to noise and outliers. Specularities and shadows for example bring outliers in the linear system since they are not predicted by the Lambertian model. For these reasons, more images should be used leading to a least squares solution of the system below for each pixel:

$$\begin{pmatrix} l_{1,x} & l_{1,y} & l_{1,z} \\ l_{2,x} & l_{2,y} & l_{2,z} \\ l_{3,x} & l_{3,y} & l_{3,z} \\ & \dots & \end{pmatrix} \begin{pmatrix} an_x \\ an_y \\ an_z \end{pmatrix} = \begin{pmatrix} I_1 \\ I_2 \\ I_3 \\ \dots \end{pmatrix}$$

Once this system is solved the lenght of the vector gives a while its direction gives n. With the normals at hand, we can solve for the color albedos with another least squares solution. In fact this solution has the following closed form:

$$k_c = \frac{\sum_i I_{i,c} < l_i, n >}{\sum_i (< l_i, n >)^2}$$

where i is the image and c stands for the color channel (r,g,b).

As noted above, capturing RGBNs comes down to taking photographs of an object under varied lighting conditions and then solving a linear system. Even though capture is simple, if RGBNs are to claim their place as first class geometry representation, there must be a variety of ways to process and edit them. Toler-Franklin et al. adapted many Stylized Depiction algorithms for working with RGBNs. In this process they also adapted signal processing and segmentation. In the next section we review and extend their work on RGBN analysis and filtering.

1.4 Analysis and Filtering

RGBNs are extended images, since they include geometry information. As such very efficient geometry processing tools can be developed. These tools take advantage of working with an image. Uniform sampling simplifies neighbourhoods and provides memory locality.

We begin this section by reviewing the work of [11] on gaussian and bilateral filters for RGBNs. Followed by a new method which allows us to use any kernel for filtering. The authors of [11] adapted segmentation methods and also developed estimates for differential properties of the surface given by an RGBN.

1.4.1 Filtering

The simple way to filter an RGBN would be to consider each channel of a 6D (color + normal) image separately and convolve it with a kernel. Since the resulting normals would not have unit norm, a normalization step would follow the filtering. The problem with naive filtering is that due to foreshortening the area of each pixel will be underestimated by $\cos\theta$. The angle θ being between the normal and the viewing direction. Each pixel should be weighted by this factor. To simplify the analysis we assume a constant viewing direction as in the case of a far away viewer. In this case the viewing direction is the z direction and $\cos\theta = n_z$. This analysis means we should replace the normal vector (n_x, n_y, n_z) by $(n_x/n_z, n_y/n_z, 1)$, which we call foresight corrected normal. Smoothing in this representation is now a linear operation. Any filter with no gain in the DC frequency will preserve the z component as 1 and its true value can be recovered by normalization. Gaussian filtering in particular satisfies this property. Among other applications, smoothing will be used to remove noise and for multiscale analysis.

1.4.2 Bilateral Filtering

A Gaussian filter is an extremely useful tool, but it has one drawback: edges are not preserved. The problem is although pixels near an edge are very close spatially their

colors are not correlated. The idea of bilateral filtering [12] is to take into account not only domain weights (distance) but also range weights (color similarity). This results in the following non-linear filter:

$$C^{-1} \int_{\mathbb{R}^2} \mathbf{f}(\xi) k_x(\xi - x) k_c(\mathbf{f}(\xi) - \mathbf{f}(x)) \mathrm{d}\xi$$

, where

$$C = \int_{\mathbb{R}^2} k_x(\xi - x) k_c(\mathbf{f}(\xi) - \mathbf{f}(x)) \mathrm{d}\xi$$

is a normalization constant to preserve the DC component.

The k_c kernel measures the color similarity (distance) between pixels. Most often, the k_x and k_c kernels are Gaussian kernels with, respectively, σ_x and σ_c as standard deviations. Small values of σ_c will better preserve edges.

In [11], bilateral filtering was extended to RGBNs, range filtering will be performed not only in color but also in the normal channel. The filtering equation generalizes to the following:

$$C^{-1} \int_{\mathbb{R}^2} \mathbf{f}(\xi) k_x(\xi - x) k_c(\mathbf{f}(\xi) - \mathbf{f}(x)) k_n(\mathbf{n}(\xi) - \mathbf{n}(x)) \mathrm{d}\xi$$

, where

$$C = \int_{\mathbb{R}^2} k_x(\xi - x) k_c(\mathbf{f}(\xi) - \mathbf{f}(x)) k_n(\mathbf{n}(\xi) - \mathbf{n}(x)) \mathrm{d}\xi$$

$\mathbf{f}$ and $\mathbf{n}$ are vector valued functions, respectively, color and normal.

Large values of σ_c and σ_n will give kernel values close to 1, as such, the filtering will only be affected by the domain filter with kernel k_x. When σ_n is small normal discontinuities (creases) will be well preserved. If both are small then normal edges and color edges will enhance each other.

1.4.3 General Filtering

In subsection 1.4.1 we reviewed the work of [11] where filters with zero gain in the DC frequency were shown to work linearly in the foresight corrected normal representation $(n_x/n_z, n_y/n_z, 1)$. In this section we investigate the problem of filtering normals with any kernel.

Figure 1.3: Bilateral filtering preserves image edges.

Assumptions that lead to a height map

We are interested in establishing the equivalence between a filter in a height map representation of a surface and its normal representation. We do not want to obtain a height map explicitly, but it is a good abstraction to develop filters for normals. Assume our surface is given by $z = z(x, y)$. We can write the normal field as a function of its derivatives $z_x(x, y), z_y(x, y)$. We use the following surface parametrization $\phi(x, y) = (x, y, z(x, y))$ whose tangent vectors are $\phi_x(x, y) = (1, 0, z_x), \phi_y(x, y) = (0, 1, z_y)$. Since the normal is orthogonal to the tangent plane we get:

$$N(x, y) = \frac{\phi_x \times \phi_y}{|\phi_x \times \phi_y|} = \frac{(-z_x, -z_y, 1)}{\sqrt{z_x^2 + z_y^2 + 1}}$$

The above formula lets us convert from z_x, z_y to $N(x, y)$. In fact the foresight correction scheme proposed in subsection 1.4.1 is the reverse process. Given a unit normal $N(x, y) = (n_1, n_2, n_3)$ we can use the above formula to show that:

$$-n_1/n_3 = -\frac{\frac{-z_x}{\sqrt{z_x^2+z_y^2+1}}}{\frac{1}{\sqrt{z_x^2+z_y^2+1}}} = z_x$$

and

$$-n_2/n_3 = -\frac{\frac{-z_y}{\sqrt{z_x^2+z_y^2+1}}}{\frac{1}{\sqrt{z_x^2+z_y^2+1}}} = z_y$$

This can be done as long as $n_3 \neq 0$ in which case there is no height map that represents this surface. What we have shown is a one-to-one mapping between the

$N(x,y)$ and $z_x(x,y), z_y(x,y)$, as such, we can work with one or the other indiscriminately.

$$\begin{array}{ccc} z & \xrightarrow{*g} & z*g \\ \uparrow & & \downarrow \\ z_x, z_y & \xrightarrow{(1),*g} & (z*g)_x, (z*g)_y \\ \uparrow & & \downarrow \\ N^z & \xrightarrow{?} & N^{z*g} \end{array}$$

So we are looking for a filtering algorithm that takes the normals of a height map N^z and produces the normals of the filtered height map N^{z*g}. Reconstructing a surface from its normals is not a very stable process, but conceptually we can go from N^z to z_x, z_y and then to z itself. We proceed by convolving z with a kernel g. With this new surface at hand, we can simply differentiate and take the vector product to obtain N^{z*g}. The only flaw is that we would like to avoid reconstruction. Fortunately, the arrow (1) above will provide a shortcut since:

$$(z*g)_x = \frac{\partial(z*g)}{\partial x} = \frac{\partial z}{\partial x} * g = (z_x * g)$$

Equivalently for y,

$$(z*g)_y = \frac{\partial(z*g)}{\partial y} = \frac{\partial z}{\partial y} * g = (z_y * g)$$

What this means is that when the normal is foresight corrected $(-z_x, -z_y, 1)$ we can convolve it with any kernel. Since we are only interested in filtering z_x, z_y, we can simply set the third component to 1, whether the kernel would preserve it or not. After filtering we simply normalize the vector to get a unit normal back.

This general filter procedure is clearly equivalent to the filter described in subsection 1.4.1 when a DC preserving kernel is used, even though we arrived at it through an entirely different argument.

Having developed this filtering framework, we now discuss some applications. Gaussian Filtering was discussed above, in fact, low-pass filtering would have a similar functionality. We are also interested in Sharpen Filters to enhance detail,

that is an all-pass filter + high-pass filter. The problem with sharpening is that by enhancing high-frequencies we also enhance noise. A simple solution is using an Edge Enhancement Filter using all-pass + band-pass.

All of the above filters preserve the DC frequency. Filters that do not have this property can be useful for editing normals. For example we might be interested in extracting a normal texture from an RGBN, in this case we look for eliminating the low-frequencies related to shape and retaining the high-frequencies related to texture. A Difference of Gaussians or a Laplacian of Gaussian provide simple band-pass filters. By themselves they would produce surfaces of no interest, yet they can used for edge enhancement as described above. Another application is simply scaling the surface represented by the normals generating shallow surfaces. Any filter can be regarded as a combination of scaling with a DC preserving filter.

1.4.4 Derivative, Curvature Estimates

An RGBN extends an image because it also includes geometry information. As such, a natural question to be posed is how can we estimate the differential geometry of our surface. Since we already have the normal at hand, the only first order entity we want to calculate are tangent vectors. To obtain them, we regard the RGBN surface as being parametrized in the image cartesian plane:

$$\begin{aligned} \phi(x,y) &= (x,y,z(x,y)) \\ u = \frac{\partial \phi}{\partial x} &= (1,0,z_x) = (1,0,-\frac{n_x}{n_z}) \\ v = \frac{\partial \phi}{\partial y} &= (0,1,z_y) = (0,1,-\frac{n_y}{n_z}) \end{aligned}$$

The first fundamental tensor, which can be used to calculate lenght and areas over the surface, is defined by:

$$I = \begin{pmatrix} u \cdot u & u \cdot v \\ v \cdot u & v \cdot v \end{pmatrix}$$

The second fundamental tensor is defined to be:

$$II = \begin{pmatrix} D_u n & D_v n \end{pmatrix} = \begin{pmatrix} \frac{\partial n}{\partial u} \cdot u & \frac{\partial n}{\partial v} \cdot u \\ \frac{\partial n}{\partial u} \cdot v & \frac{\partial n}{\partial v} \cdot v \end{pmatrix}$$

Knowing the I and II, we can calculate the mean curvature, gaussian curvature, principal curvatures and directions. Since u and v defined above project to unit orthogonal vectors $(1,0)$, $(0,1)$ in the image plane, we can use central finite differences to estimate the derivatives of the surface normal:

$$\frac{\partial n(x,y)}{\partial u} = \frac{1}{2}(n(x+1,y) - n(x-1,y))$$
$$\frac{\partial n(x,y)}{\partial v} = \frac{1}{2}(n(x,y+1) - n(x,y-1))$$

1.4.5 Segmentation

Segmenting RGBN produces better results than segmenting color images. The normal channel provides aditional information frequently uncorrelated with color. With RGBNs it is also possible to segment objects based solely on geometry.

In [11], the authors adapted the graph-partitioning algorithm of Felzenszwalb and Huttenlocher [13] to RGBNs. Their algorithm works bottom-up, each pixel is a vertex in a graph and starts in each own region, while edges encode neighboorhood similarity. The algorithm proceeds by merging similar regions. Although thy use a greedy criteria for merging, the resulting segmentation satisfies some global properties. Their algorithm is fast $O(m \log n)$ and is suited to interactive applications.

To adapt this algorithm to RGBN segmentation, all that is needed is a measure of neighbouring pixel similarity. We experimented with the following:

$$w(v_i, v_j) = g(|c(v_i) - c(v_j)|, |n(v_i) - n(v_j)|)$$

where g can be the min, max or a linear combination of parameter α of the differences.

By using $\alpha = 0$ or 1 we can segment using only color or normal information, respectively. It should also be clear that among all these configurations, max pro-

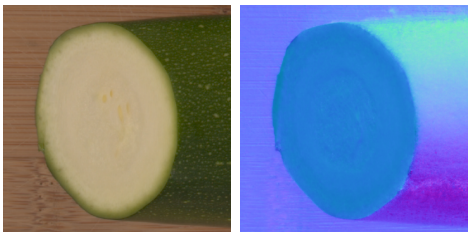

Figure 1.4: In between the white face of the cucumber and its surface the normal is discontinous. This defines a crease line.

duces the finest segmentation while *min* produces the coarsest. A bilateral filter should be used prior to segmentation to remove noise while preserving edges.

In Textureshop [14], an alternative RGBN segmentation method is proposed. It uses only the information from the normal channel. First every pixel in put in its own patch. Merging of adjacent patches proceeds when an energy functional falls below a given threshold.

1.5 Local Operators

1.5.1 Creating Features

Before defining local operators, we must answer what kind of surface modifications we are interested in. We begin by discussion different features usually found in surfaces and their mathematical descriptions. We usually work with continuous surfaces (C^0). These surfaces may or may not have continuous derivatives (C^1). We say a surface point belongs to a crease, if the surface's normal (derivative) is discontinous at that point (Figure 1.4). Creases are first order features. We can also define second order features related to surface curvature. We say a surface point belogs to a ridge, if it is a local maxima of the principal curvature in the direction of its corresponding principal direction. Equivalently we define valleys as the local minima (Figure 1.5).

It is very important to distinguish features across scales. Very large features related to low-frequencies should be edited by setting the model at a compatible

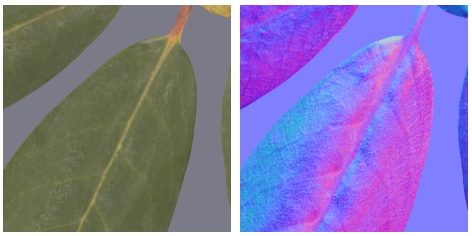

Figure 1.5: The continuation of the stem (yellow) in the leaf is a valley since curvature is a local minimum when searching perpendicular to the stem.

resolution. Very small details (high-frequency) should be edited on a high resolution version of our model. While this would be the ideal way of editing we do not yet have an RGBN multiresolution scheme suited for editing, which we leave as future work. Currently we edit a fixed scale model as will be detailed in the next section.

Up to now, we have discussed isolated features. It is common to find many features clustered together in a region forming a texture. For this type of editing, we resort to texture synthesis techniques (Section 1.6).

1.5.2 Local Editing

All the tools developed in this section work on local features (small scale). To support these operations, we first use bilateral filtering on the normal image and obtain a low resolution version of the normals (Figure 1.6). We refer to this smooth normals as base normals. These normals are useful for defining a local tangent space where normal editing will take place. The extension of this framework for editing normal maps on arbitrary meshes should follow naturally by assuming the base normals to be the mesh normals.

Since normals are mathematical objects living in S^2, any normal operation can be seen as a function $f : S^2 \to S^2$. We chose to model this functions as rotations. Using rotations it is easy to work on the local tangent plane as well as interpolate operations. In what follows, we review some results regarding Rodrigues Formula, more details can be found in [15].

Among the many ways of representing a rotation, the most intuitive is by defining

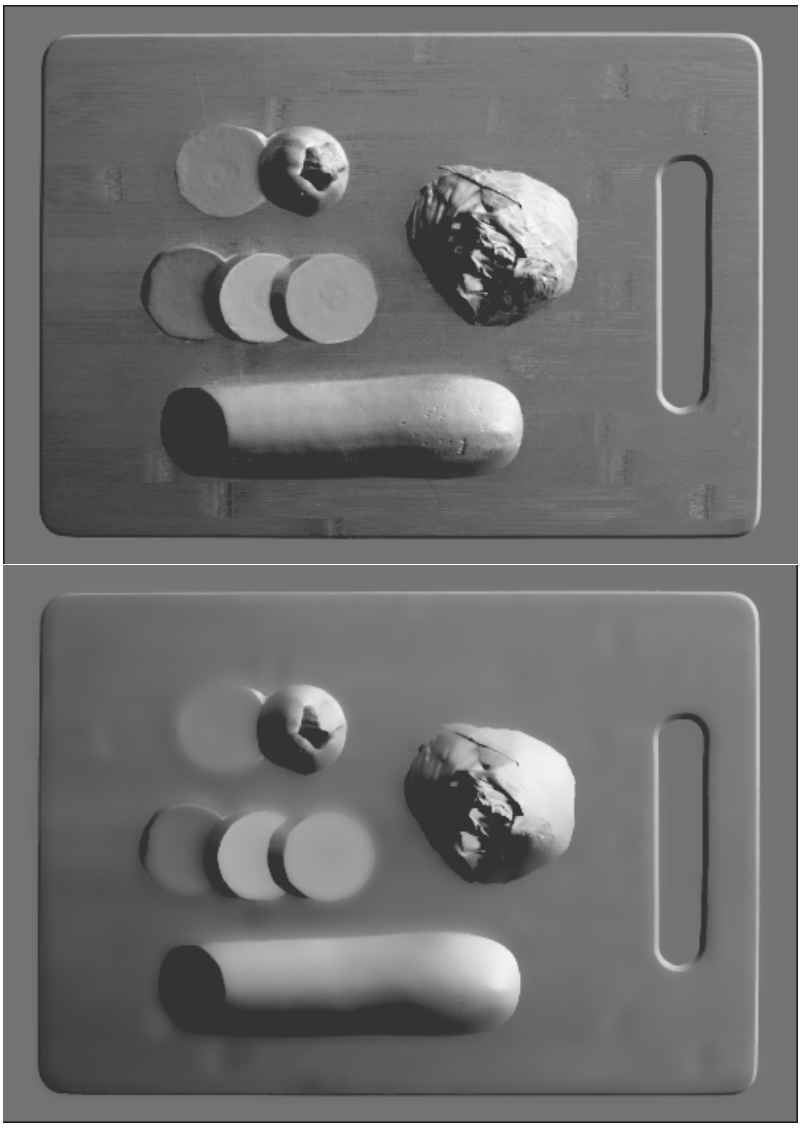

Figure 1.6: The original normal image is smoothed with bilateral filtering producing the base normals, a coarse scale version of the surface.

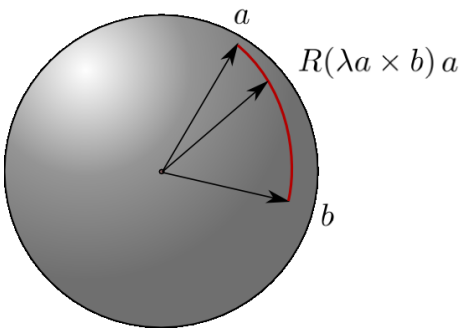

Figure 1.7: $R(\lambda a \times b)a$ is a parametrization of the great arc connecting a and b.

an axis of rotation r (unit vector) and an angle of rotation α in radians. Both of these quantities can be specified by only one vector $w = \alpha r$. The calculations are more elegant by instead representing the rotation as $w = \sin(\alpha)r$. Rodrigues Formula provides an easy way to convert between this representation and a 3D rotation matrix:

$$R(w) = cI + (1 - c)rr^t + sr^\Lambda,$$

where $s = (||w||^2)^{\frac{1}{2}} = \sin\alpha, c = (1 - ||w||^2)^{\frac{1}{2}} = \cos\alpha, sr = w$, and

$$r^\Lambda = \begin{pmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{pmatrix}$$

One important property of rodrigues formula is that given n_i, n_j unit vectors:

$$R(n_i \times n_j)n_i = n_j$$

Interpolation is also simple since $R(\lambda a \times b)$ and $R(a \times b)$ are both rotations around the same axis , the only difference being the angle (Figure 1.7).

Our local operations are transformations in a local coordinate system defined by the base normal vector n. We still have one degree of freedom for the basis of the tangent plane which will be specified by an orientation vector o. The unit vector o is in the xy plane see Figure 1.8. We define the tangent vector $u = o - proj_n(o)$

and $v = u \times n$. The final system is defined by u, v, n.

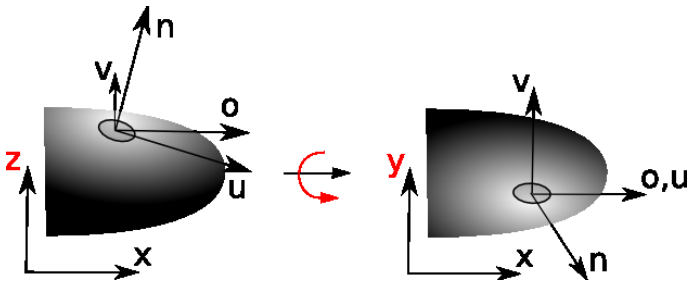

Figure 1.8: Local coordinate system on the surface.

Editing the normal starts by specifying a desired normal vector b, usually with coordinates in the local system. We could simply replace the existing normal a with b. We call this replace mode. Another option is to blend a and b, which we call blend mode. Blending can be very useful for fading smoothly the effect of the new normals as distance from the edited area increases (see subsection 1.5.4).

This strategy fails when b has a negative z value (Figure 1.9). In this case we actually need to replace b by a saturated vector k, defined as the maximum displacement along the great arc which has positive z values. As such we can either replace a with k or blend them. Notice that since k is on a great arc it depends not only on b but also on a.

A normal in the xy plane corresponds to a discontinous height field ($z_x \to \infty$). To correct this problem, we introduce a user-defined parameter ξ meaning the minimum allowed angle a normal makes with the xy plane. By saturating whenever necessary, we guarantee that the normals correspond to a coherent height field.

calculation of the saturated vector Why dont we simply interpolate vectors linearly and then normalize?

1.5.3 Stamp

The first local operator we propose is inserting a small RGBN (stamp) in a neighbourhood of a point (Figure 1.10). The stamp can be an entire RGBN itself or

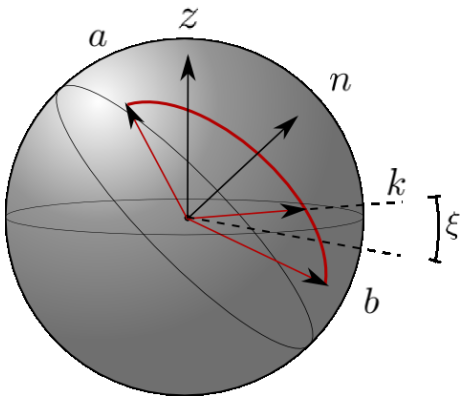

Figure 1.9: Interpolation between the original normal a and the assigned normal b might result in vectors with negative z values. To preserve RGBN coherency we calculate a saturated vector k where interpolation must stop.

extracted from one. In the last case, we must eliminate the lower frequencies through filtering when creating the stamp.

Each normal of the stamp is interpreted as being in the tangent plane of our surface while editing. So the new normal b is simply the stamp's normal transformed from the local coordinate system. To define this system, we simply take $o = (1, 0, 0)$ where o is the orientation vector defined above. Other vectors in the xy plane would cause the stamp to be rotated.

1.5.4 Pen

The pen operator is used for creating line features. The normals will be transformed to correspond to a custom height profile along the path drawn by the user. In Figure 1.11, some useful profiles can be seen, which allow the user to create bumps, creases or scratches on the surface. Custom profiles can be specified using grayscale height images.

The first step is to establish coordinates in a tubular neighbourhood of the path (Figure 1.13). This coordinates define a mapping $(s, t) = f(x, y)$ between a canonical representation of the height profile and the tubular neighbourhood. The $s(x, y)$ coordinate can be regarded as a radial displacement from the path, while the $t(x, y)$

Figure 1.10: The leaf was used as a stamp and added as detail to the soldier's skirt. Notice how the results of each stamp are different, depending on the base normals.

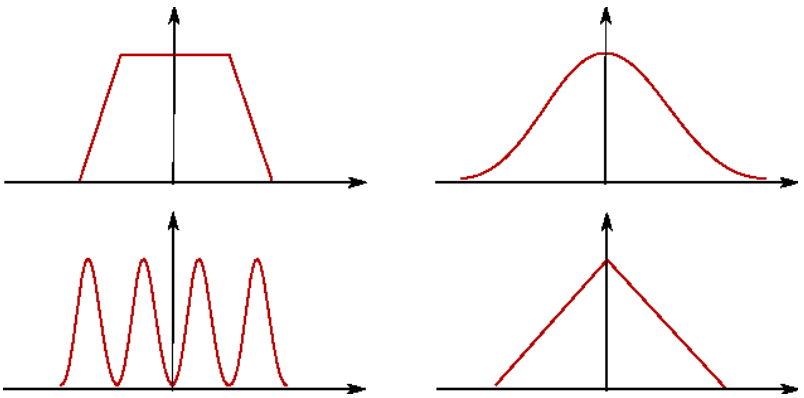

Figure 1.11: Custom height profiles along a path are transfered to the normals with the pen operator.

Figure 1.12: This cucumber was edited using the four profiles above. Can you match them?

coordinate as a displacement along the path. In the pen operator the profile is independent of the t coordinate, which simplifies the calculations. Formally $s(x, y)$ is taken as the distance from the path.

The sampling of the path drawn by the user is interpreted as a piecewise linear continous curve (C^0). To avoid slow distance field calculations, we use a geometrical calculation using point to segment distance functions (Figure **??**). The t coordinate is not calculated explicitly but is interpreted as being an arc-length distance from the beginning of the path up to the point of projection. This corresponds to the exact distance field if we assume a small tubular neighbourhood and a path with no self-intersections. Even when this hypothesis fails, intuitive results are still obtained. The $f(x, y)$ mapping defined using the distance field inherits properties of the distance field itself (continuity). Its derivative is not continuos on the medial axis of the path. This discontinous derivative will lead to discontinous normals (creases) in the final surface, this characteristic might or not be desirable. Instead of the distance field approach, one could define the mapping $f(x, y)$ by extracting control points based on the input path and building a spline surface. This mapping can be built C^1 everywhere.

After editing the normals does the resulting normal field correspond to a surface? If not, how should we reconstruct an approximate surface from its normals? We leave the second question for future research. By now, we investigate conditions that will

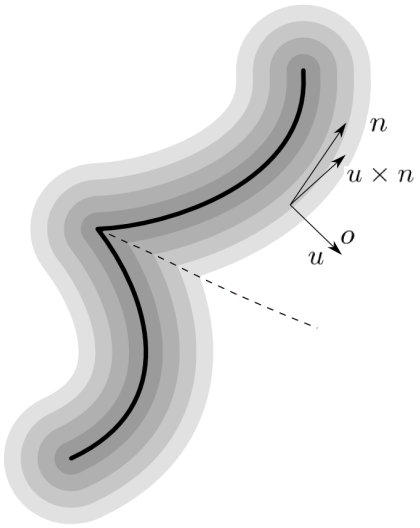

Figure 1.13: The orientation vector is determined in a tubular neighbourhood of the user-drawn path. It is the gradient of a distance-field. Notice that this vector field is discontinous over the path's medial axis (dotted line).

allow us to answer: yes, it is a surface. We refer to these fields as integrable normal fields.

We begin by studying the action of the pen operator on a planar region.

Let $h(u, v)$ be the applied deformation. In the case of the pen operator, $h(u, v)$ does not depend on v but the following proof is general. The differential 1-form defined by:

$$\omega = A(u, v)du + B(u, v)dv$$

where

$$A(u, v) = \frac{\partial h(u, v)}{\partial u}, B(u, v) = \frac{\partial h(u, v)}{\partial v}$$

is trivially an exact form since $dh = \omega$. We want to show that the integral of the 1-form induced by the deformation on the plane is "invariant".

In this simpler case of plane deformations, the original normal used for defining a local frame is always $n = (0, 0, 1)$. As such, our local coordinate system defined by the gradient of the distance function is $(\frac{\partial U}{\partial x}, \frac{\partial U}{\partial y}), (\frac{\partial V}{\partial x}, \frac{\partial V}{\partial y})$.

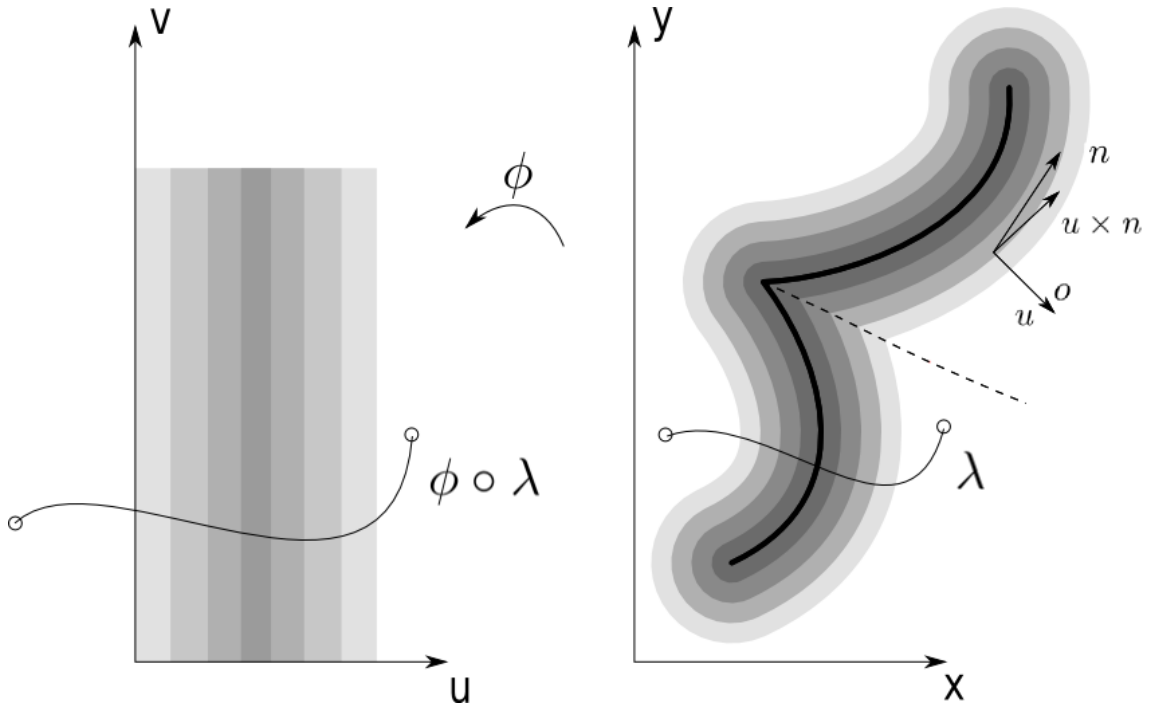

Figure 1.14:

The resulting derivatives (gradient) after the deformation is applied to the plane will be:

$$A(\phi(x,y))(\frac{\partial U}{\partial x},\frac{\partial U}{\partial y}) + B(\phi(x,y))(\frac{\partial V}{\partial x},\frac{\partial V}{\partial y})$$

$$= (A(\phi(x,y))\frac{\partial U}{\partial x} + B(\phi(x,y))\frac{\partial V}{\partial x}, A(\phi(x,y))\frac{\partial U}{\partial y} + B(\phi(x,y))\frac{\partial V}{\partial y})$$

. We must now recognize this induced form as the pullback $\phi^*(w)$. This leads us to an important result:

$$\int_\lambda (A(\phi(x,y))\frac{\partial U}{\partial x} + B(\phi(x,y))\frac{\partial V}{\partial x})dx + (A(\phi(x,y))\frac{\partial U}{\partial y} + B(\phi(x,y))\frac{\partial V}{\partial y})dy = \int_\lambda \phi^*(\omega) = \int_{\phi\circ\lambda} \omega$$

.

To show that $\phi^*(w)$ is an exact form, we may show that the integral of $\phi^*(w)$ over a closed path λ is 0. This holds since w is an exact form:

$$\int_\lambda \phi^*(\omega) = \int_{\phi\circ\lambda} \omega = 0$$

.

As demonstrated above, integrability will be preserved after an operation. This means there is a height field $f(x, y)$ with the given derivatives.

In fact, this is not enough to preserve integrability after many operations. To support this claim, we give a counter-example inspired by this drawing of Escher (Figure **??**). In the image below, many deformations were made. The paths are radial and the height profile was a step function as shown in Figure 1.15. Each edit is a step up, just like in Escher's infinite stair. This example shows how even though the deformation was locally coherent, it might not be globally coherent. This happens since not all closed paths λ above are contained in the edited area.

To fix this issue, we ask one more property of the derivatives of our deformation function $h(u, v)$:

$$\exists R, |r_1|, |r_2| > R \Rightarrow \int_{r_1}^{r_2} \omega = 0$$

This implies that for "far away" points no height level change will be introduced. This can also be seen as having a window of low-frequencies not allowed in these derivatives. They must oscilate. By the invariance of the integral above, the edited normals will also integrate to 0.

To extend this result to surfaces other than the plane, we argue that if the base normals are sufficiently smooth, it can be locally considered a plane. This is justifiable since the base normals only have low-frequencies and the deformation only has higher frequencies. These information are in different levels of detail. This leads to a selection criteria for the parameter R above. The higher the value of R, the smoother the base normals need to be.

There is still one other problem. The resulting normals could be away from the camera or even making multiple turns along a path λ on the surface. Since our deformation comes from a height profile that satisfies the frequency separation criteria above, these problems will not happen. Yet, there are situations where the above hipothesis are not properly satisfied, for example if R is not chosen small enough. In these cases, the normal saturation procedure defined in the subsection 1.5.2 will guarantee our results.

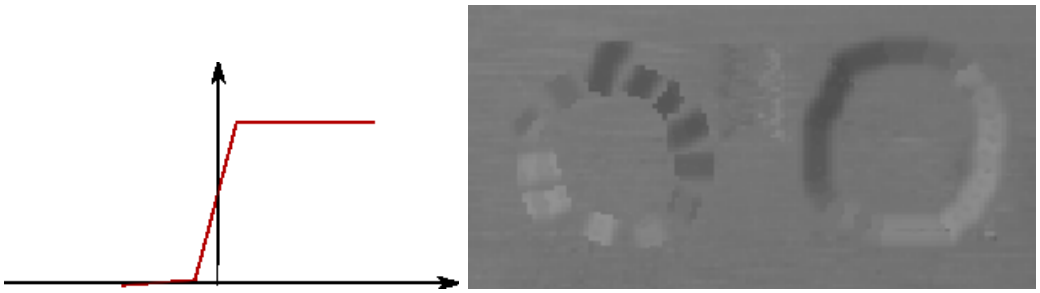

Figure 1.15: On the left a sequence of small deformations were made on a planar surface, each edit is a step up. There is no surface with this given normals. On the right a surface bump is shown. It is interesting to notice how the shading of an impossible object ressembles a rotated version of a real one.

1.5.5 Local Filtering

We have also developed a local filtering operator. It behaves just like the general filtering procedure described above, but only normals in a small neighbourhood of a pixel are edited. The shape and radius of this neighbourhood can vary. In Figure 1.16, we show an edited version of , both smoothing and enhancing the normals. Notice how enhancing the signal also enhances noise.

1.5.6 Experiments with integrability error

A vector field $f = (u(x, y), v(x, y))$ that is the gradient of a height function is curl-free. That is $E(x, y) = |(\nabla \times f)(x, y)| = |\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}| = 0$. As such, this value can be used to measure the deviation of an edited normal field from a true surface's normal.

The following figures show $E(x, y)$ plotted in a non-linear grayscale. Notice how the pen operator actually reduces the integrability error, since the edited surface is smoother than the original one.

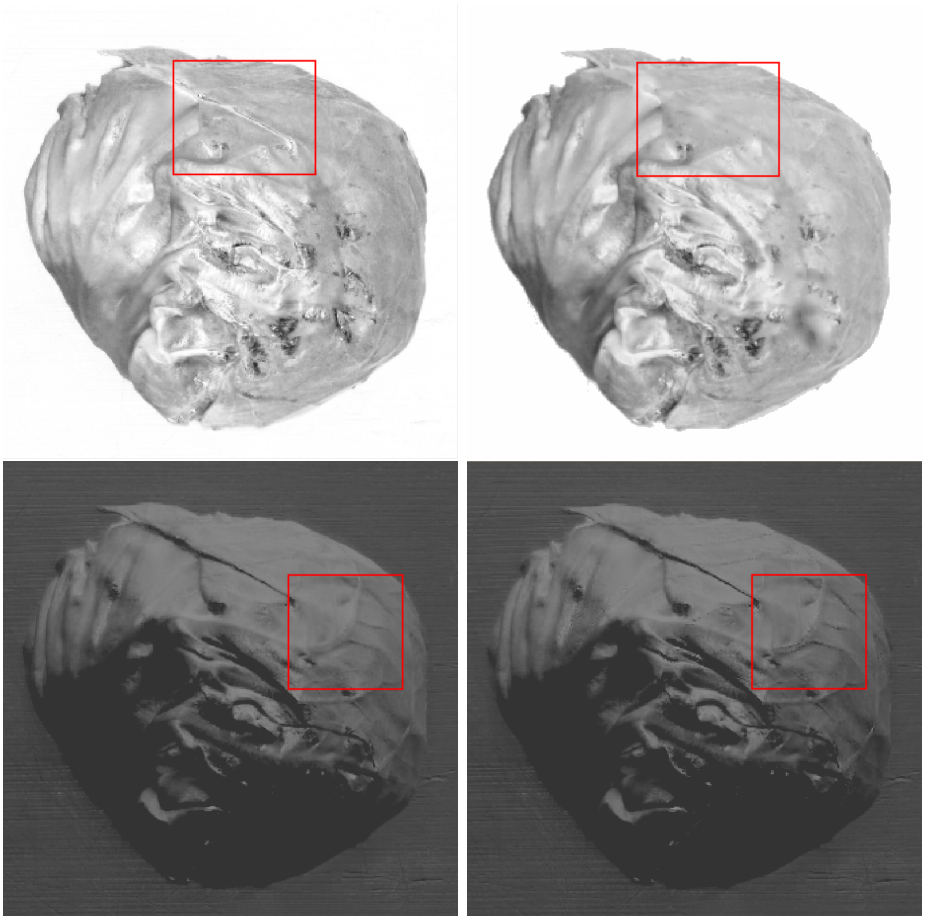

Figure 1.16: The detail of the image on the upper right was smoothed. The lower left one was enhanced.

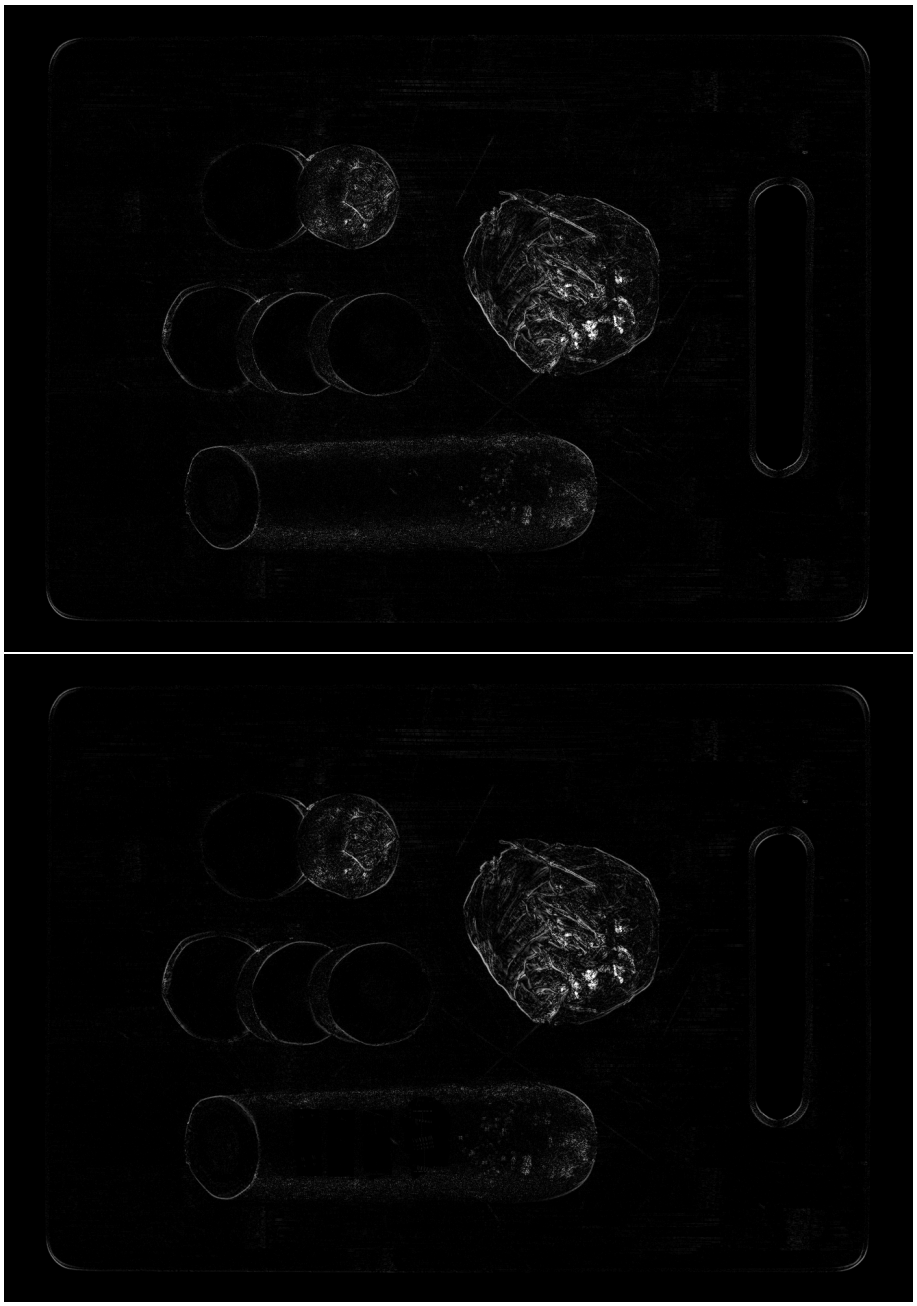

Figure 1.17: The first image shows the integrability analysis before editing. The second shows it after the editing session of Figure 1.12.

1.6 Texture Synthesis

1.6.1 Region Editing

Editing regions by individually addressing local changes can be painstaking to artists. For this reason this process needs to be made automatic. One has two simple options: procedural methods and texture from example. Procedural methods are very powerful but are difficult to control. Two algorithm classes are common. In the first one, a material would be synthesized in a volume (marble in a cube, for example) and then a surface would be cut from this volume. When working with RGBNs we do not have positions, so we neither have a surface nor a volume. It would be complex to adapt these algorithms. The second class does procedural synthesis directly on the surface, usually with local decisions. These methods might be adaptable to RGBNs, but we chose not to investigate this topic.

Texture from Example methods on the other hand are very intuitive. They take as input only a small sample of the desired texture and then reproduce it in a large region. As the input consists of a sample, it is very easy to gather in the real world or design small artificial textures.

We chose to approach editing details in a large region as a texture from example problem. The edited region can be defined using the segmentation tools described previously. Editing a large region will never be as fast as real-time local editing. Even though, as we are dealing with an interactive software it is a requirement to synthesize texture fast.

The previous section described local editing. In all operators presented there was the need to establish a coordinate system in the edited region. To extend this approach to large regions we are faced with a parametrization problem but building a good parametrization is a complicated problem by itself. So it is desirable that the texture synthesis procedure requires no parametrization.

The trivial solution to the Texture from Example problem is to tile the sample but this approach has two shortcomes. First seams (border between adjancent tiles) will be easy to notice (Figure 1.18). Second, the result will be periodical. The

human mind is trained at finding patterns and for most textures this periodicities will not seam natural.

The trick to solve this problem is that if we acquire a good sample of our desired texture, it should contain all necessary information for us to reproduce the original synthesis process. This means whatever the synthesis process was, be it natural or human-made, we can reproduce it from a sample. This argument is made formal by regarding texture as a stationary stochastic process.

A stochastic process $F_{(x,y)}$ is a family of random variables taking values in a set S. This family is indexed by a set X. In the case of 2D texture synthesis, S is the set of colors while $X = R^2$. This means for each $(x_1, y_1) \in R^2$, $F_{(x_1,y_1)}(c)$ is a random variable in the space of colors.

Since $F_{(x_1,y_1)}(c)$ is random variable, we can look at the joint probability density function $F_{(x_1,y_1),(x_2,y_2),...,(x_k,y_k)}(c_1, c_2, ..., c_k)$. In the case where $k = 2$ and $(x_1, y_1), (x_2, y_2)$ are neighbouring pixels, this function could answer how likely we are to find a black and a white pixel side by side. A stochastic process is called stationary if:

$$F_{(x_1,y_1),(x_2,y_2),..,(x_k,y_k)} = F_{(x_1+\triangle x,y_1+\triangle y),..,(x_k+\triangle x,y_k+\triangle y)}$$

$$\forall(\triangle\ x, \triangle\ y) \in R^2$$

This means the joint probability function is space invariant, that is it depends only on the relative position of the points being observed.

We also require independence:

$$\exists d, \forall |x_1 - x_2| > d \rightarrow F_{x_1,x_2}(c_1, c_2) = F_{x_1}(c_1)F_{x_2}(c_2)$$

This means if we acquire a sample texture big enough from this process, we have access to all its statistics, like its expected value, autocorrelation and moments.

This also implies that the quality of the sample will deeply influence the quality of the results. Since many ocurring textures are stationary, texture from example synthesis works well in practice.

In [16], the authors generalize the above restrictions on textures. They ask

Figure 1.18: Even for simple examples tiling introduces seams and periodicities.

for quasi-stationarity, meaning that while joint probability distribution is not independent of position, it changes slowly. The texture sample cannot answer how it changes, which is usually left to the user to information extracted from the surface (curvature). The authors show smooth morphings between different texture samples.

Texture from example synthesis can be classified by two means: synthesis domain and synthesis step. From the step point of view, there are pixel-based methods where new pixels are generated one at a time. In [17], the authors search for a best match between the neighbourhood of the already synthesized texture and neighbourhoods in the texture sample. There are also patch-based methods [18] [14] where the algorithm step consists of iteratively synthesizing small regions at a time. This approach leads to a direct transfer of local statistics, but it has the drawback that seams between patches need to be handled.

As to the domain classification, texture synthesis can be parametric, volume-based or manifold-based. Volume-based methods will synthesize texture in an R^n domain, usually R^2 or R^3. Parametric synthesis will target more general manifolds by working in the parametric domain (usually planar). There are also manifold methods which are non-parametric [19] [20]. These methods work directly in the manifold representation (usually a mesh) and being non-parametric usually means only local decisions can be made.

We do not have an explicit surface, only its normals. For this reason, pixel-based synthesis together with non-parametric manifold methods are a best fit with RGBNs. In the next subsection, we review Jump maps based synthesis, our texture engine choice for handling RGBN editing.

1.6.2 Jumpmaps

The first example-based synthesis [17], [18] would exhaustively search the input for a best match. This search would be done for each pixel or patch being synthesized. In this section we review [3] where Zelinka splits texture synthesis in two phases: analysis and synthesis. Note that in previous methods analysis was being done online

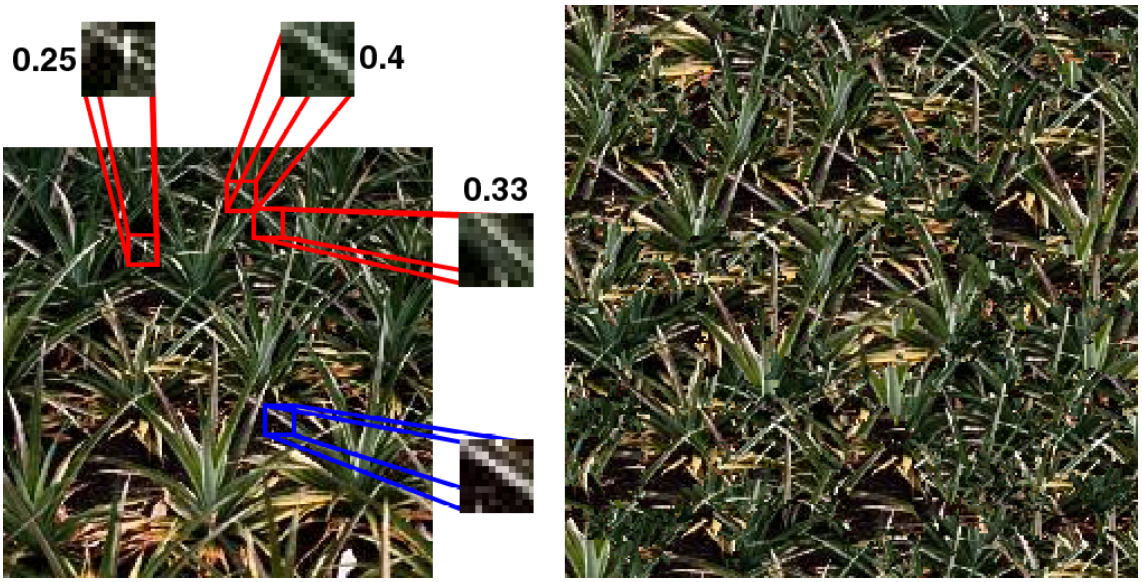

Figure 1.19: Similarities between neighborhoods are encoded into a Jump Map. Figure taken from [3].

and would use an already synthesized neighborhood as a query. Zelinka's insight is that while this neighborhood is only known at synthesis time it will resemble an existing one in the texture example. As such we can precalculate the similarities between all neighborhoods in the input texture. They store this information in the Jump Map data structure, which holds for each example pixel a list of jumps to other pixels that have similar neighborhoods (Figure 1.19). The Jump Maps synthesis is pixel-based and runs in real-time, requirements of our system. It also generalizes to surfaces in a non-parametric way as desired for RGBNs.

During synthesis, the output image is traversed and pixels are sequentially copied from the example image. Eventually the input image border will be reached and this simpe process is unable to continue. That is when the Jump maps come in hand. As the border approaches a jump is randomly selected from the Jump Map. As such, there is an infinite amount of texture to be copied.

The analysis phase is cast as a nearest-neighbor problem in a neighborhood space. Each pixel's MxM neighborhood is encoded in a M^2 feature vector, using the L_2 norm for comparisons. Since small masks won't represent the texture appropriately

we are easily lead to a high dimensional NN problem which would be too slow to handle directly. Instead, the authors use an approximate nearest neighbors (ANN) data structure [21] that allows very fast queries with errors of 10%. Second, they reduce the size of the feature vector using Principal Component Analysis, retaining 97% of the variation.

Another important optimization is to first build a multiresolution pyramid of the example texture. The resulting feature vector will be concatenated from the feature vectors of the same neighborhood in different pyramid levels. This is unintuitive since the new vector is bigger. Yet, PCA performs much better in this bigger vector and the result after PCA compression is much smaller. It is interesting to notice, that most textures are very tolerant to these approximations, meaning that synthesized results are not very affected.

Synthesis is a little more complex than described. Each pixel is synthesized receiving the color of a corresponding pixel in the example texture. When borders are reached, jumps are taken leading to a different location in the input image. In fact, even far from borders, there is a probability p that a jump will be made instead of proceeding synthesis to the neighbouring pixel. This p parameter controls the expected number of pixels between jumps and thus controls the expected size of texture patches fully copied from the example image. Note that this is a Bernoulli process.

Nothing was said about the order in which pixels of the output image are synthesized. This order is in fact very important leading to differente results. Scan-line and serpentine traversals introduce much directionality in the final result. Zelinka found that following Hilbert curves produces results without bias.

A generalization to surface was also proposed in [3]. Instead of iterating over the output pixels, the new algorithm iterates over mesh vertices. There is now the need to map a displacement on the mesh to a displacement in texture space. This can be done using only a supplied orientation field and scale defined on the surface [3]. With this mapping at hand, it is now simple to adapt the previous synthesis algorithm: iterating over vertices induces a walk in texture space from which texture

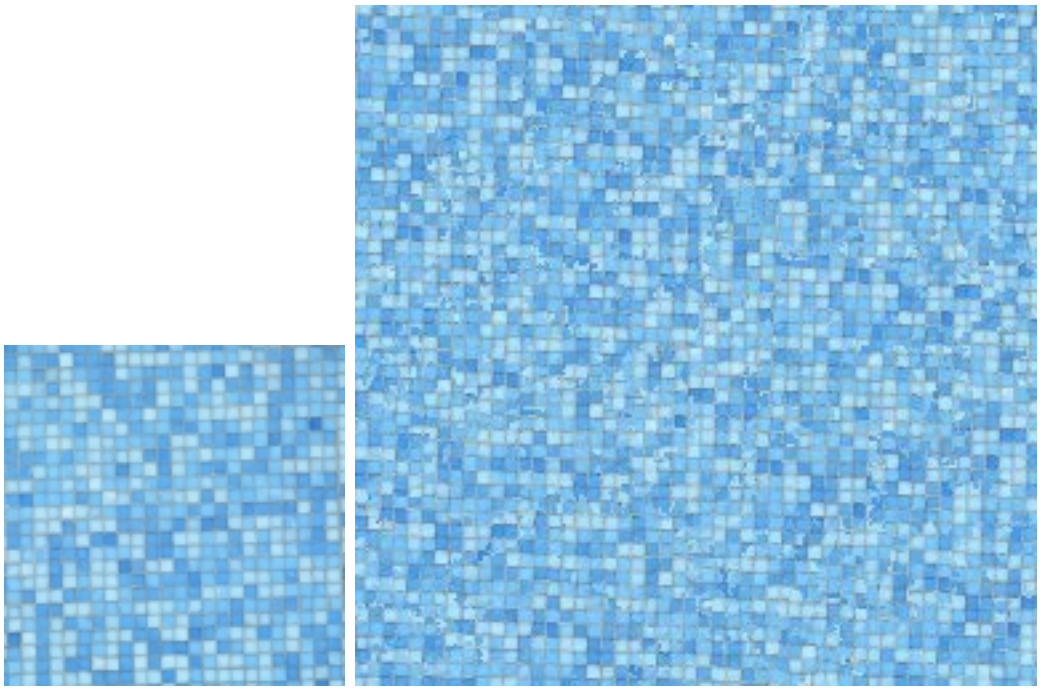

Figure 1.20: Jump Map synthesis results.

can be copied. Note that synthesis order (vertices order) is still very relevant, we refer the reader to the original work. In the previous subsection, we were looking for a texture synthesis method that was non-parametric, pixel-based and very fast. JumpMaps synthesis fits all our requirements.

From here on, we present our observations.

As the authors notice, jump-map synthesis works best for stochastic textures and weak structured textures. While some perfect results can be obtained with highly structured textures (Figure 1.23), these are exceptions rather than rule. In Figure 1.27, we see very similar neighborhoods, but if a jump is taken between them offsets will be introduced in the synthesized bricks 1.24. This problem happened because a small neighborhood mask was used for analysis. Notice how a bigger mask size improves the results (Figure 1.27).

Another problem with structured textures is that sometimes structure is very clear in our human eyes, while not so much in the example (Figure **??**). One simple way out is to tile the sample image few times creating a new sample (Figure **??**)

Figure 1.21: Jump Map synthesis results.

Figure 1.22: Jump Map synthesis results.

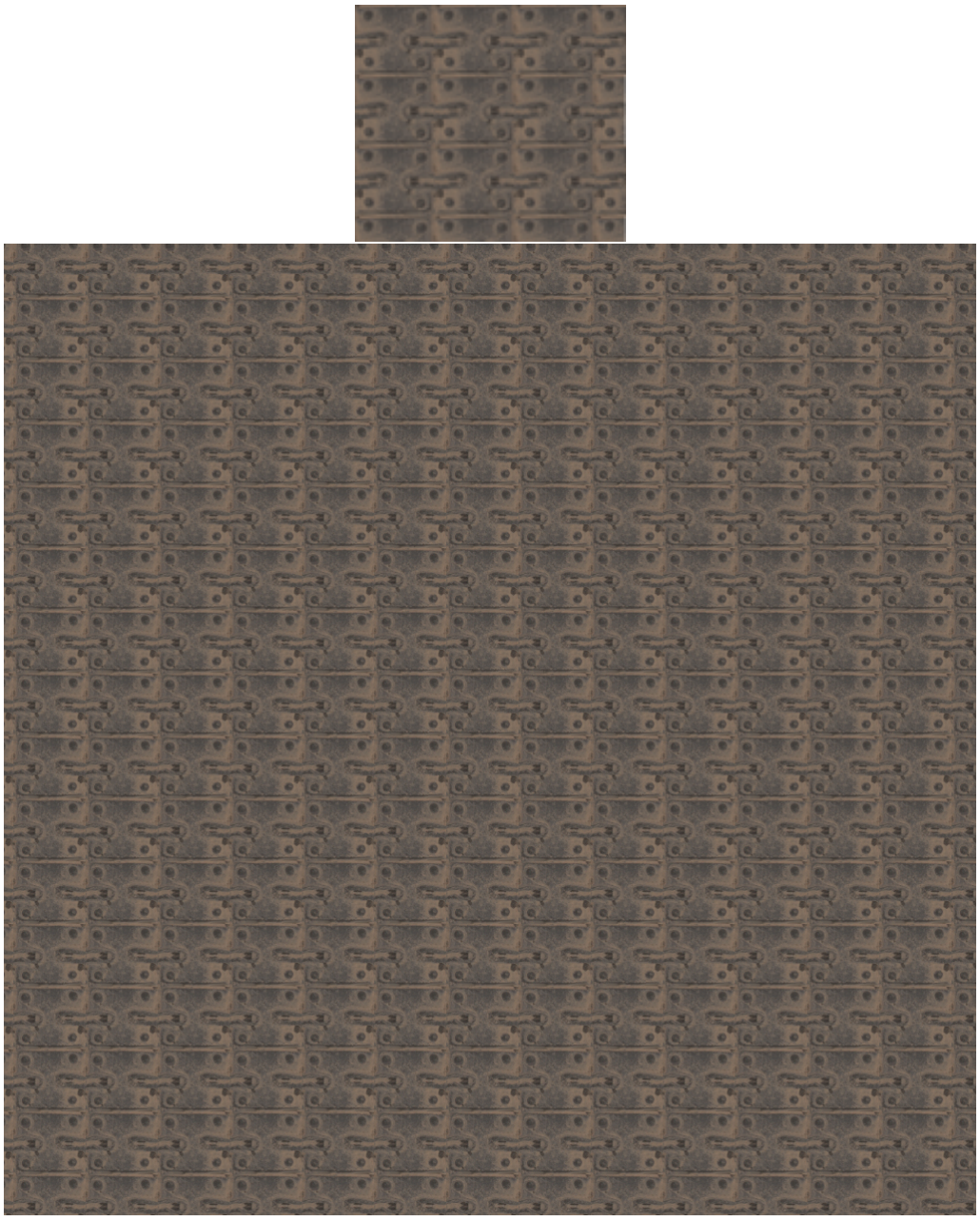

Figure 1.23: Synthesis of a highly structured texture.

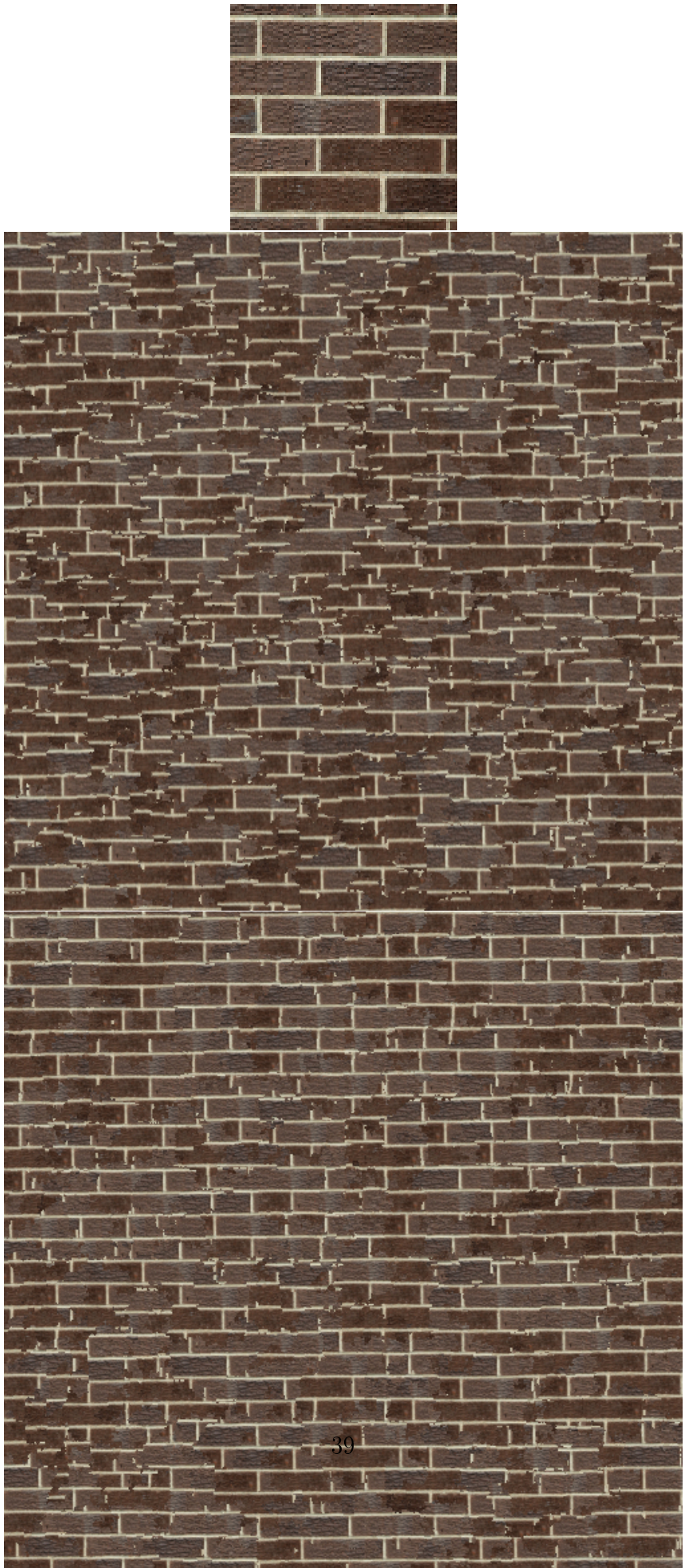

Figure 1.25: The three neighborhoods are very similar, but if a jump is taken between them offsets will be introduced in the synthesized bricks, as can be seen in Figure 1.24.

and then proceeding with synthesis. It would have been simple to tile the entire output image in the first place, but this solution would not work for synthesis on surfaces or RGBNs.

While increasing the mask size improves results, it also improves analysis time. This means only samples of smaller resolution will be processed in a fixed time budget. We notice that it is possible to use different samples (different resolutions) for analysis and synthesis, virtually allowing larger masks. Small samples can be used for fast analysis, while detailed samples can be used for quality synthesis (Figure 1.29). This is possible but requires floating-point texture offsets, that is relative offsets that are independent of resolution. This floating-point offsets are a natural consequence of the Jump-maps on surface extension. Notice that a full analysis on the 500x500 image would simply take too long (Figure 1.29). In a sense, you will be telling the computer that the large scale structure features are more important than the small scale texture. As the results show, quality results can still be obtained.

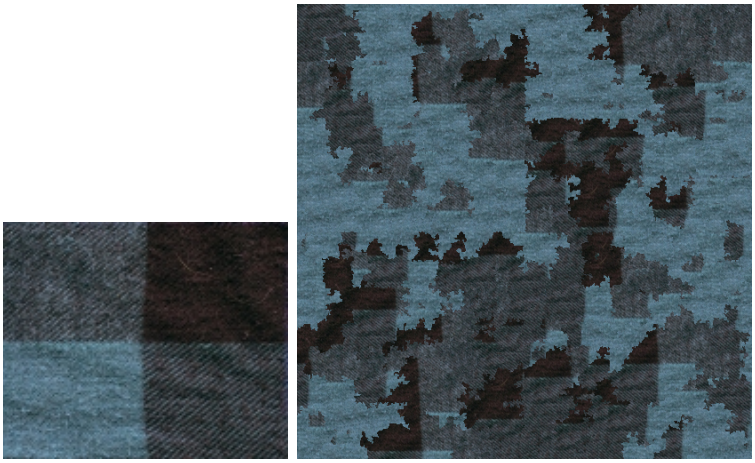

Figure 1.26: Structure is not clear from the sample leading to disatreous results.

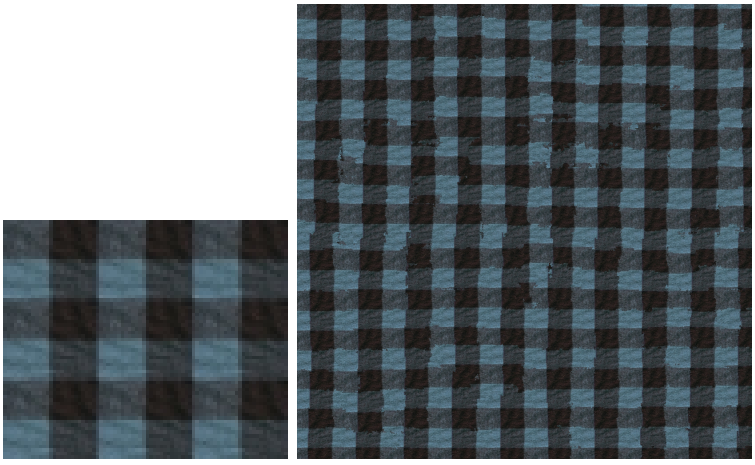

Figure 1.27: Structure is more evident from the sample leading to good results.

1.6.3 Texture Synthesis on RGBNs

In Textureshop [14], the authors transfered normals between images. Poisson image editing [22] was used to merge normals seamlessly (in each channel? followed by normalization?), a process which does not lead to a resulting conservative normal field. Our normal transfer builds on our framework of normal editing and LOD separation (base normals) yielding approximately conservative fields as detailed below. Since Textureshop does not involve base normals in the process, their results appear to be floating, even though they are seamless. In simple scenarios, perspective corrections would do the trick but not in more involved ones.

While the synthesis of normals on RGBNs has not been properly explored, in [23] the authors adapted jumpmap-based color synthesis to normal images. Notice the difference between the signal being synthesized and the domain on which it is synthesized. Their work follows more directly from jumpmaps on surfaces where a texture offset is calculated based on the mesh displacement. As for normal images, the intuition is that a unit pixel offset in the output image induces a displacement in the represented surface, which in turns induces an offset in texture space. For this purpose, a scale s_p and an angle of orientation θ_p are assigned by the user. This parameters may be global or may vary smoothly over the surface [16]. By approximating the surface locally by a plane orthogonal to the normal n, we can obtain the displacement d_t in texture space as a function of the displacement d_i in output image space:

$$d_t = \frac{s_p |d_i|}{|n_z|} \frac{p}{|p|}$$

where

$$p = d_i - < n, d_i > n$$

is the projection of d_i onto the surface.

Since d_t lies on the surface, all that is left is to represent d_t in the texture space basis. This basis is build by rotating by θ_p the tangent plane basis (see subsection 1.4.4).

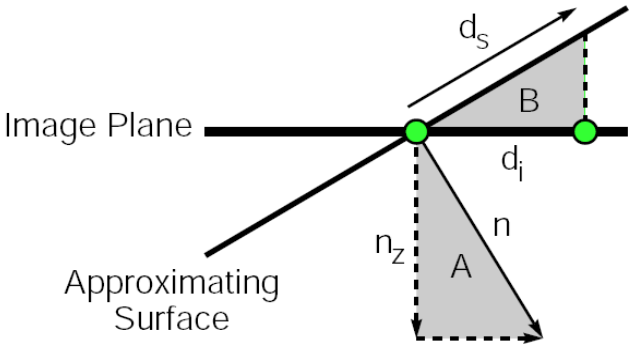

Figure 1.29: Texture offsets can be calculated from pixel offsets in the output image.

Now that we understand how to synthesize colors on normal images, we can tackle the problem we were interested in the first place: how to synthesize normals on normal images. We split this discussion in analysis and synthesis as we are using JumpMaps.

In the analysis phase, we take a normal map sample representing our desired texture and we are asked to build a jumpmap for it. The basic primitive in jumpmap analysis is comparing neighborhoods. In the color setting, the L_2 metric was used, which is easily extended from comparison of individual pixels (RGB) to comparison of big feature vectors representing neighborhoods. Remember that using L_2 metric for feature vectors was very important for subsequent optimizations like PCA and ANN. So first we must settle on which metric to use for comparing normals, followed by normal neighborhoods. Finally, how can this norm be optimized for the NN problem?

Mathematically speaking, each feature vector is a collection of normals $n \in S^2$. As such the feature vector V:

$$v \in V = S^2 \times S^2 \times ... \times S^2$$

the cartesian product of a number of unit spheres.

This leads to the natural definition of a metric in V, inherited from any S^2 metric d:

$$d_V(v_1, v_2) = \sum d(v_{1,k}, v_{2,k}), v_1, v_2 \in V$$

where $v_{1,k}, v_{2,k}$ are the projections on the kth unit sphere.

We review three different metrics that could be used as the S^2 metric d above: L^2, geodesic and dot product based.

Since normals are mathematical objects in S^2, it would be natural to use a geodesic metric on the unit sphere to compare two normals n_1, n_2. This could be achieved by calculating $d_1(n_1, n_2) = cos^{-1}(< n_1, n_2 >)$. While it would be easy to adapt the $O(N^2)$ solution to the NN problem, it is very hard to adapt either PCA or ANN, the fundamental algorithms that allow building jump-maps in reasonable time. We would either need an ANN solution in non-Euclidean spaces or use manifold learning techniques to embed our neighborhoods in an euclidean space. Both of these solutions would be complex as the cos^{-1} is a non-linear function.

The linear dot product based metric $d_2(n_1, n_2) = 1 - < n_1, n_2 >$ would definitely simplify the above matters with linearity. But d_2 falls far from the geodesic metric (Figure 1.30). While scaling would yield a good approximation to the geodesic metric, it would still fail for small values. As we are interested in NN problems, these small distances are the ones we are most interested in.

Figure 1.30 also shows that the L_2 metric is a very good approximation for the geodesic distance, being specially good for small distances since their derivatives at zero agree. Since we already have good texture analysis methods for the L^2 metric and it agrees with geodesic metric for most of the values we care about in the NN problem, we understant this is the best solution for comparing normal neighborhoods.

Note that previous arguments allows us to use plain Jump-maps to synthesize normal maps.

We look for synthesis in subsets of RGBNs, these could be defined by user or automated segmentation. Since this regions may have very complex shapes and topologies, it would be more complicated to use a synthesis order based on Hilbert curves. We used a depth-first search (DFS) using a 4-connected pixel neighborhood.

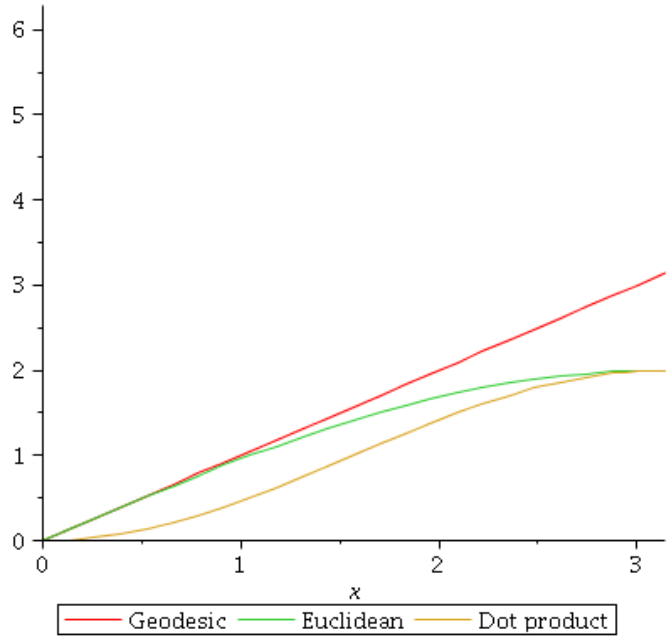

Figure 1.30: Different S^2 metrics behaviour as function of the angle between the normals.

The naive DFS algorithm produces unpleasant results with a directionality bias. A simple alternative is to use a DFS and random shuffle each pixels neighborhoods. This breaks directionality and generates results (Figure 1.34) almost as good as Hilbert traversal ones, even in plane images.

We synthesize normals on RGBNs as described above. Usually the normal sample encodes normal details of a planar surface, so the appropriate interpretation for the synthesized normals is that they are in tangent space. To produce the final RGBN these synthesized normals need to be mixed with the existing normals. The process is analogous to what was done for local editing: a lower resolution version of the normals (base normals) are used to define a tangent plane allowing us to obtain the final RGBN (Figures 1.35 and 1.37).

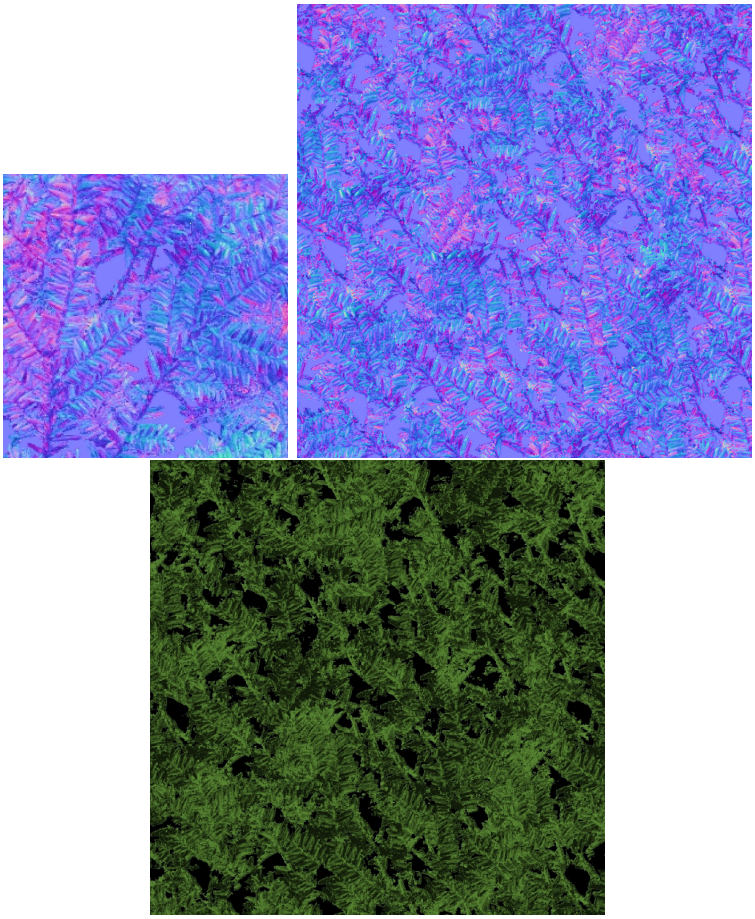

Figure 1.31: The leaves were replicated generating a normal map. On the right, a shaded version of the synthesized map.

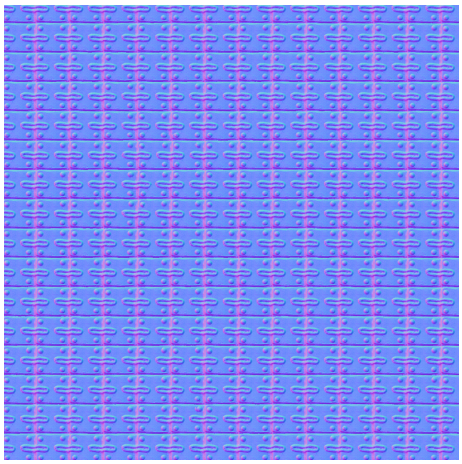

Figure 1.32: Some structured normal maps can be generated.

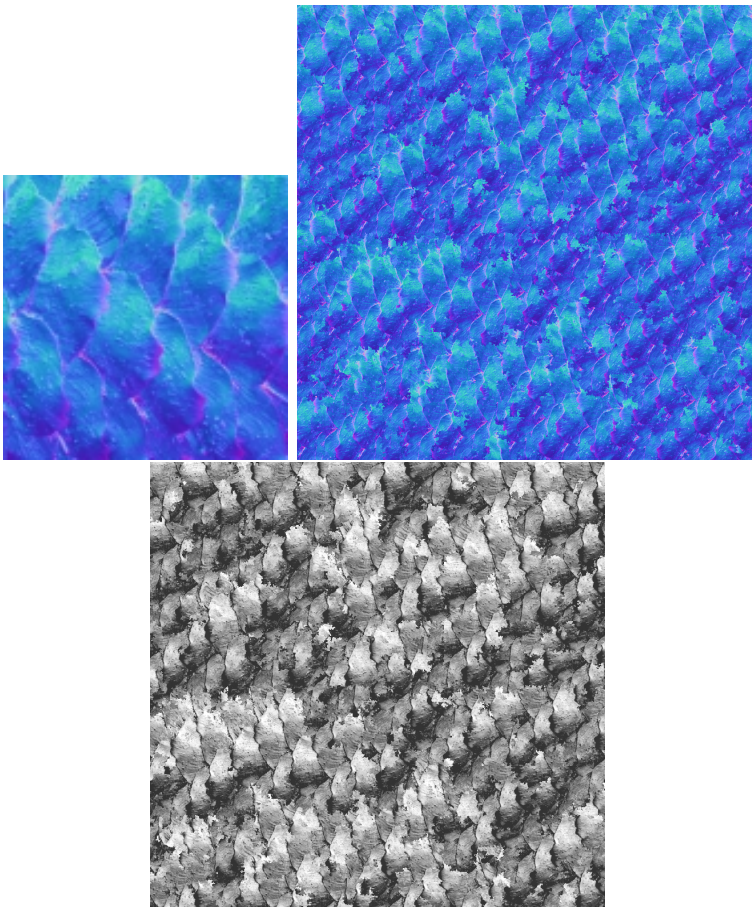

Figure 1.33: The tree scales were synthesized on a normal map. On the right, shaded normals.

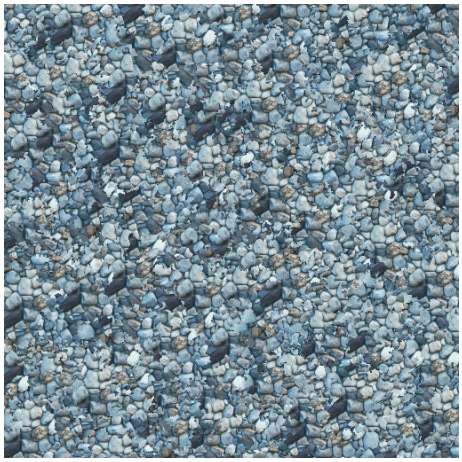

Figure 1.34: Depth-first traversal with random shuffled pixel neighborhoods generates good results.

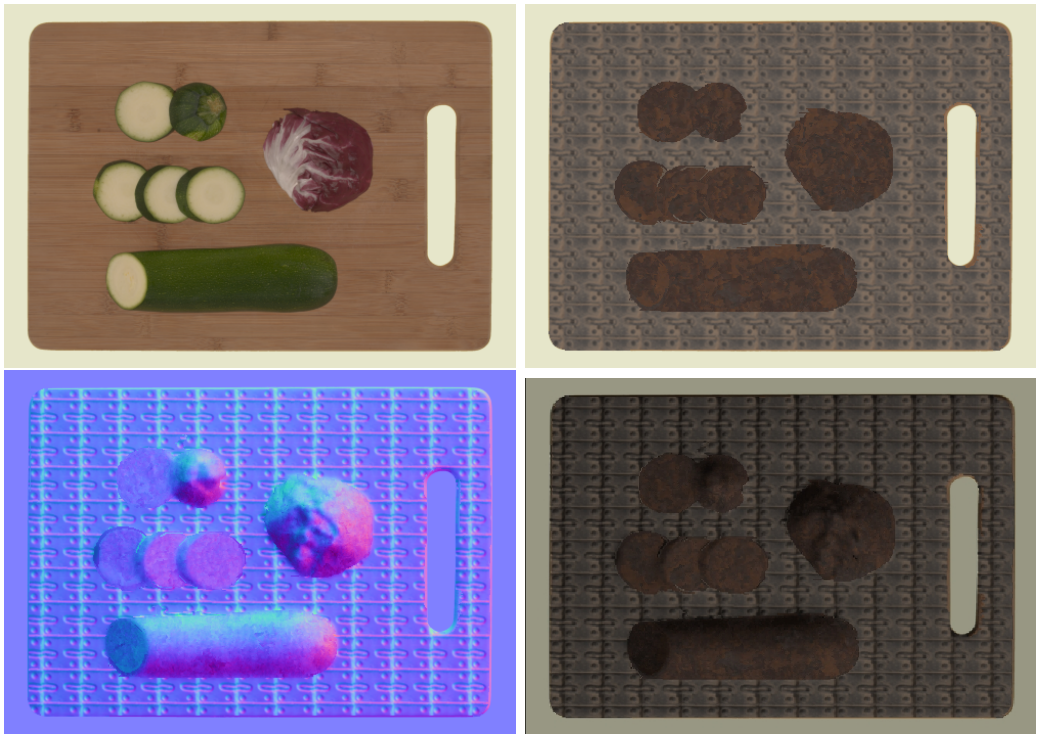

Figure 1.35: Automatic segmentation followed by texture synthesis is a powerful tool for adding detail and material replacement. The vegetables were transformed into rust metal.

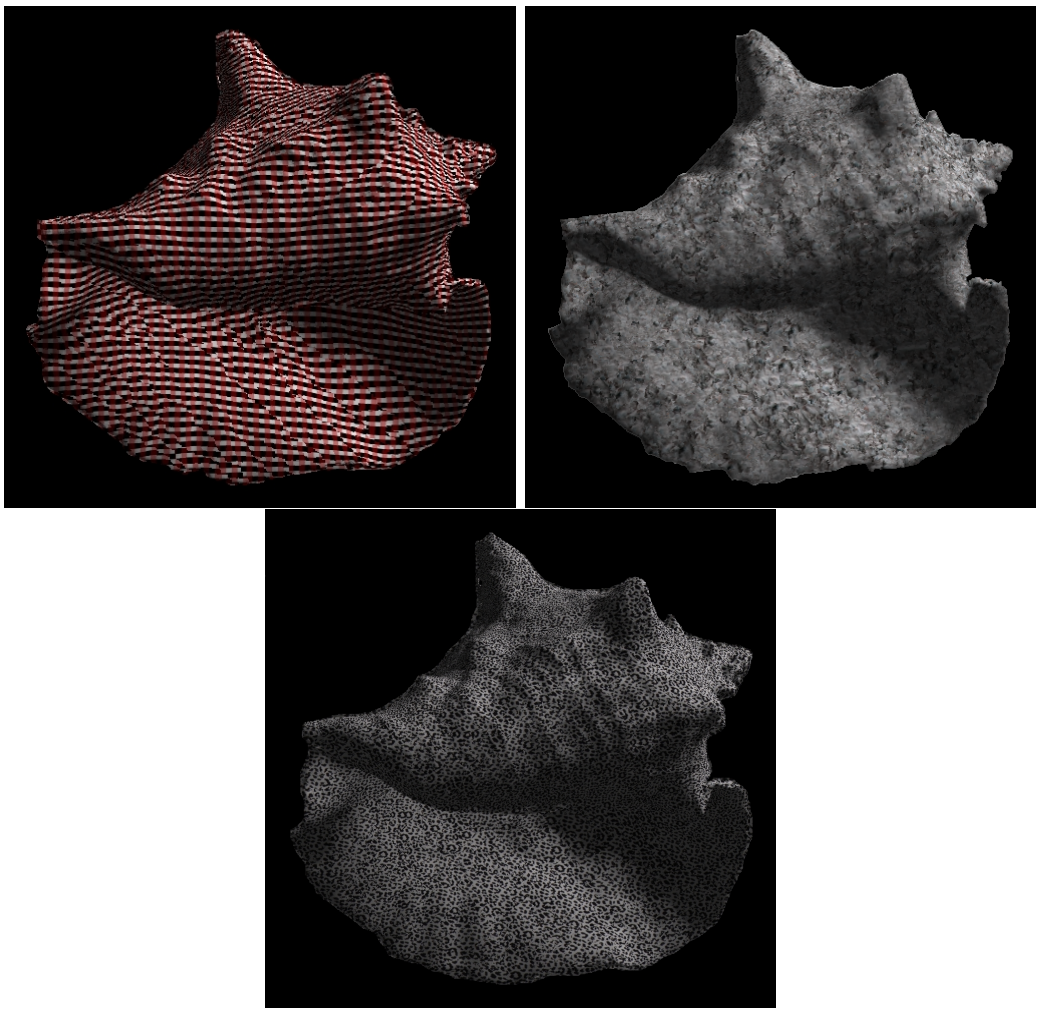

Figure 1.36: Different textures synthesized on a shell.

Figure 1.37: Rust normals synthesized on a shell.

1.7 Conclusions and Future Work

RGBNs are one of the simplest ways to enhance color images with geometric information. They are very easy to capture requiring inexpesive hardware and achieve much better resolution than 3D scans.

A greater number of users should be able to work with RGBNs and achieve some effects that were only possible with the use of modelling tools like Maya.

While the authors of [11] have developed many analysis and rendering algorithms for RGBN and we have proposed many editing operations, there is still much to be developed.

Bibliography

[1] James F. Blinn. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, 12(3):286–292, 1978.

[2] Ignacio CastaÃśo J.M.P. van Waveren. Real-time normal map dxt compression. *Technical Report*, 2008.

[3] Steve Zelinka and Michael Garland. Jump map-based interactive texture synthesis. *ACM Trans. Graph.*, 23(4):930–962, 2004.

[4] Tongbo Chen, Michael Goesele, and Hans-Peter Seidel. Mesostructure from specularity. In *CVPR '06: Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1825–1832, Washington, DC, USA, 2006. IEEE Computer Society.

[5] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving simplification. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 115–122, New York, NY, USA, 1998. ACM.

[6] P. Cignoni, C. Montani, R. Scopigno, and C. Rocchini. A general method for preserving attribute values on simplified meshes. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 59–66, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[7] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, 1985.

[8] B. K.P. Horn. Shape from shading: A method for obtaining the shape of a smooth opaque object from one view. Technical report, Cambridge, MA, USA, 1970.

[9] Diego Nehab, Szymon Rusinkiewicz, James Davis, and Ravi Ramamoorthi. Efficiently combining positions and normals for precise 3d geometry. *ACM Trans. Graph.*, 24(3):536–543, 2005.

[10] Paul Debevec, Tim Hawkins, Chris Tchou, Haarm-Pieter Duiker, Westley Sarokin, and Mark Sagar. Acquiring the reflectance field of a human face. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 145–156, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[11] Corey Toler-Franklin, Adam Finkelstein, and Szymon Rusinkiewicz. Illustration of complex real-world objects using images with normals. In *International Symposium on Non-Photorealistic Animation and Rendering (NPAR)*, August 2007.

[12] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision*, page 839, Washington, DC, USA, 1998. IEEE Computer Society.

[13] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient graph-based image segmentation. *Int. J. Comput. Vision*, 59(2):167–181, 2004.

[14] Hui Fang and John C. Hart. Textureshop: texture synthesis as a photograph editing tool. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 354–359, New York, NY, USA, 2004. ACM.

[15] Gabriel Taubin. Linear anisotropic mesh filtering. *Technical Report RC-22213*, October 2001.

[16] Jingdan Zhang, Kun Zhou, Luiz Velho, Baining Guo, and Heung-Yeung Shum. Synthesis of progressively-variant textures on arbitrary surfaces. In *SIGGRAPH*

'03: ACM SIGGRAPH 2003 Papers, pages 295–302, New York, NY, USA, 2003. ACM.

[17] Alexei A. Efros and Thomas K. Leung. Texture synthesis by non-parametric sampling. In *ICCV '99: Proceedings of the International Conference on Computer Vision-Volume 2*, page 1033, Washington, DC, USA, 1999. IEEE Computer Society.

[18] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 341–346, New York, NY, USA, 2001. ACM.

[19] Li-Yi Wei and Marc Levoy. Texture synthesis over arbitrary manifold surfaces. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 355–360, New York, NY, USA, 2001. ACM.

[20] Greg Turk. Texture synthesis on surfaces. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 347–354, New York, NY, USA, 2001. ACM.

[21] David M. Mount and David M. Mount. Ann programming manual. Technical report, 1998.

[22] Patrick Pérez, Michel Gangnet, and Andrew Blake. Poisson image editing. *ACM Trans. Graph.*, 22(3):313–318, 2003.

[23] Steve Zelinka, Hui Fang, Michael Garland, and John C. Hart. Interactive material replacement in photographs. In *GI '05: Proceedings of Graphics Interface 2005*, pages 227–232, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2005. Canadian Human-Computer Communications Society.

1.8 Software

1.8.1 User Interface

The software is composed of three basic windows: main window, toolbar window and canvas window.

The main window contains two menu items: File and Window. In the File menu basic functionality can be accessed like New, Open, Save and Preferences. The Window menu lets the use toggle visibility of the auxiliary windows described in the following subsection.

In the toolbar window the operations described in this report can be reached. The user also selects whether editing normals or color (or both) is enabled.

The canvas window is where editing takes place just like a usual image editing software. The main visualization is a shaded RGBN in which the albedo channel is used for color and the normal channel introduces the shading. The user can also configure the light position. The ability to change the light while editing is very important since normals are usually perceived through shading.

Other visualizations are also possible (Figure 1.38). In the Window > Buffers menu, the current visualization buffer can be selected including: Shaded Image, Albedo, Shaded Normals, Color-coded Normals, Segmentation, Base-level Normals, Mean Curvature, Gaussian Curvature and Normals as Needles.

The following screenshots show the parameter window of each RGBN operation in the system.

1.8.2 Architecture

Since our system supports real-time shading of the RGBN we could not simply use OpenGL pixel operations to display the images. Instead we chose to render a textured mesh and leverage OpenGL shading capabilities. The mesh is a triangulation of the plane with two triangles for each image pixel. This allows us to specify normal and texture coordinates of each vertex at image resolution. The albedo channel is loaded into a texture. By changing this texture we can also render the other visual-

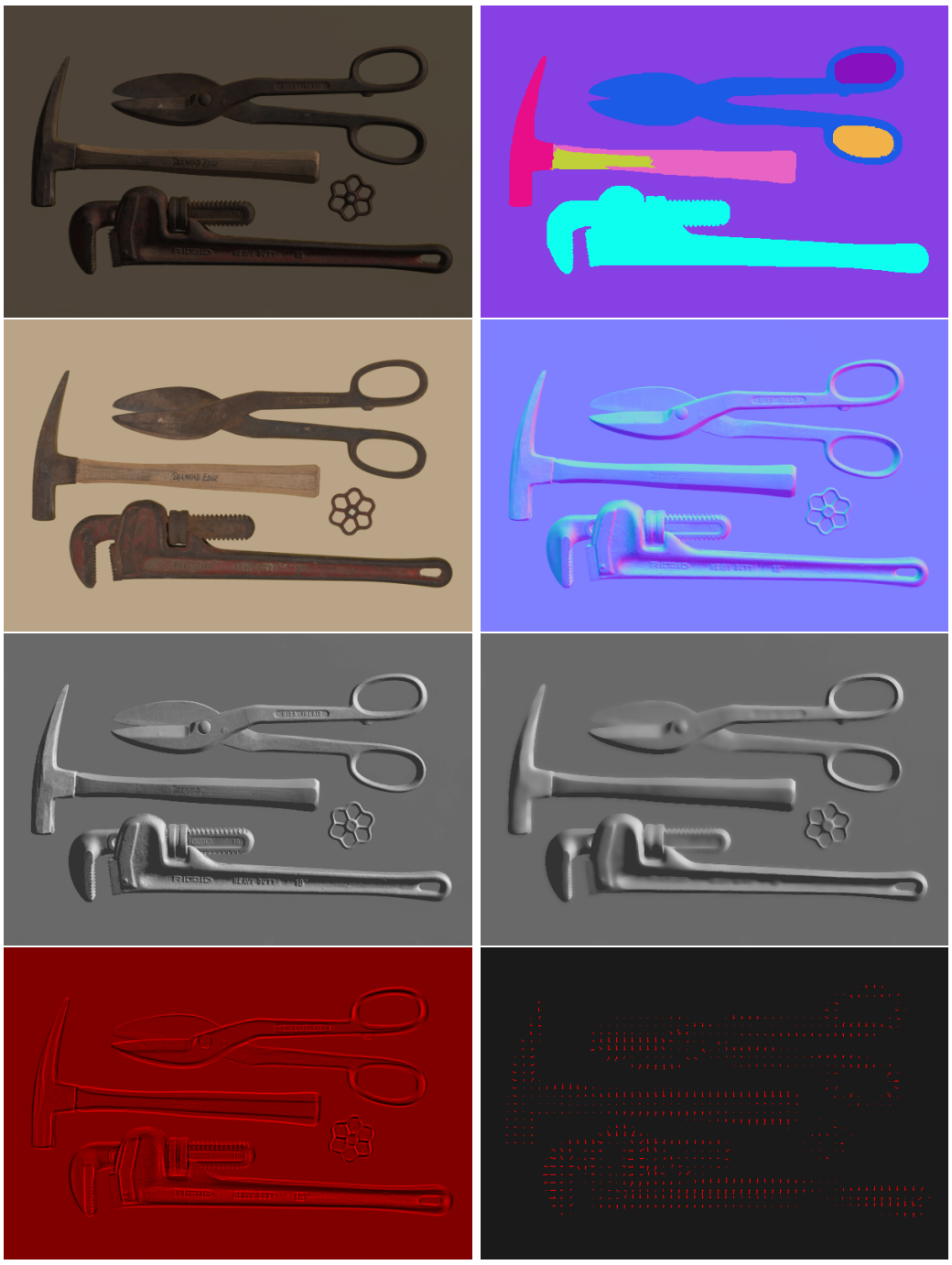

Figure 1.38: The system provides differente visualizations of an RGBN.

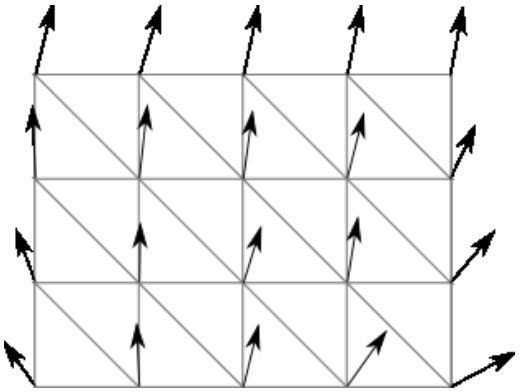

Figure 1.39: The RGBN is represented as a triangulation of the plane with two triangles for each pixel. Normals are assigned on a per-pixel basis

izations shown in the previous subsection. Notice that lighting and texturing need to be turned on and off appropriately for each buffer.

Simple operations like panning and zooming are implemented by moving the virtual camera.

This rendering architecture meets the real-time requirements of rendering high resolution RGBN without using pixel shading in the GPU. As such, it works in a broader range of graphics cards.

As described above this design has scalability issues. Current cameras can easily generate 4Mpixel pictures. As such, we would be dealing with 8 million triangles. Data transfer between the CPU and the GPU can easily become a bottleneck. To avoid this constant transfer of triangles, we use display lists. A display list allows objects to be transfered to the GPU and stored there. A simple use case would be to load an RGBN and build a display list (DL) with its corresponding mesh. At every rendered frame, we can pan or zoom and simply invoke this DL. No data transfer will take place.

During pan, zoom, relighting and even editing colors, the mesh does not change. The only color related information in the DL are the texture coordinates which are constant. To support color editing we only need to update the texture itself. As such, the DL is still valid and rendering will be very fast. The problem comes when

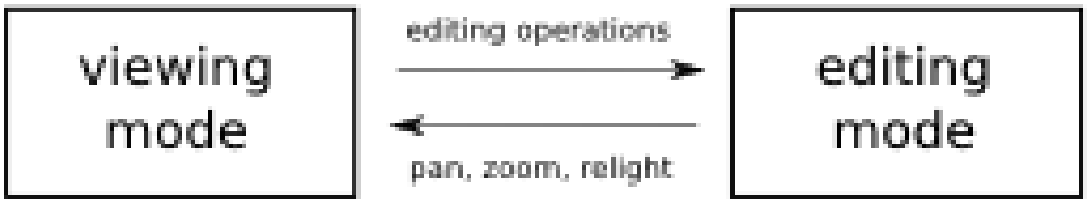

Figure 1.40: The system operates in one of two rendering modes. In editing mode the framebuffer is not cleared and only changes are rendered at each frame.

editing normals. The normals are recorded into the DL and so it is invalid if normals change during an editing session. The user must be able to see his deformations at real-time, so we just can't rebuild the DL at every change.

We chose to separate rendering into two differente modes: Viewing Mode and Editing Mode (see Figure 1.40). During Viewing Mode the normals are assumed not to change and so does the DL. We can pan, zoom and relight in Viewing Mode. When the user start editing the system is transfered to Editing Mode. During editing the DL is invalid, so no calls to render the DL are made. During any editing operation, the changed pixels are rendered on top of the previous image. This means we do not clear the frame buffer, just draw on top of it. Rendering is fast since we only render a small portion of the pixels at each time. Whenever the user pans, zooms or relights the RGBN the system automatically rebuilds the DL and goes to Viewing Mode.

All interactions required by editing operations nail down to a simple primitive of picking. In general, picking refers to detecting which object the user has selected. In our case, we are interested in which pixel was clicked.

To implement picking efficiently, we use OpenGL to render two buffers. In the framebuffer, the image seen by the user is rendered. An auxiliary buffer is used to hold the (x, y) coordinates of the RGBN pixel that was rendered into that buffer pixel. We encode this two integer values into an RGBA color value of 32 bits as supported by OpenGL. As such there are 16 bits for each coordinate.

With this auxiliary buffer at hand, picking is very simple. Screen coordinates

provided by the mouse events are used to access the buffer and the RGBA color is decoded into 2D coordinates.