INSTITUTO NACIONAL DE MATEMÁTICA PURA E APLICADA

NormalShop

Modeling surface mesostructure

Thiago Pereira

Advisor: Luiz Velho

Rio de Janeiro, February 12, 2010

NormalShop [1]

Modeling surface mesostructure

Thiago Siqueira Pereira

Thesis presented to the
Instituto Nacional de Matemática Pura e Aplicada - IMPA
for obtaining the title of
Master in Mathematics, specializing in Computer Graphics.

This work was supported by a CNPq fellowship.

Rio de Janeiro, February 12, 2010

Thesis committee:
Luiz Carlos Pacheco Rodrigues Velho (advisor) - IMPA
Luiz Henrique de Figueiredo - IMPA
Thomas Lewiner - PUC-Rio
Ricardo Guerra Marroquim - UFRJ
Paulo Cezar Pinto Carvalho (substitute) - IMPA
Diego Fernandes Nehab (reader) - Microsoft Research

[1] Shameless follower of the recent trend in graphics including Textureshop, Pointshop3D, HDR-Shop, GradientShop, ShapeShop, BRDF-Shop, Head-Shop and others.

Acknowledgements

First of all, I thank my parents for loving and supporting me in every moment of my life. All I have accomplished I owe them.

My sincere thanks to my advisor Luiz Velho. His enthusiasm and knowledge both motivated and inspired me to follow a research career in computer graphics.

I have really enjoyed my time in the stimulating environment of Visgraf. There I found many friends and many interesting discussions. My thanks to Emilio, Ives, Ilana, Sergio, Adriana, Leonardo, Djalma, Marcelo, Leandro, Francisco, Conrado, Gabriel, Rafaella, Allan, Carlos, Priscila and Bruno. An additional thanks to all whom I had the opportunity to work with. It was a great pleasure.

I am very grateful to all my professors at IMPA for showing me the beauty of mathematics and the thrill of research. Thanks to Thomas Lewiner, Luiz Henrique and Claudia Justel. Your recommendations will allow me to continue my studies abroad.

Finally, a thanks to all who helped in reviewing the text in this manuscript. A special thanks to the members of my thesis committee.

Resumo

Normais têm sido usadas em aquisição e renderização de geometria por anos. Entretanto, a modelagem com normais é limitada devido à falta de operações formalmente definidas. Nós apresentamos diversas operações formais de normais em superfícies baseadas em dois ingredientes-chave. Primeiro, uma separação do modelo em uma superfície base e detalhes, enquanto a base é representada por posições em uma malha, os detalhes só são representados através de normais em um espaço paramétrico. Nossas operações podem processar as normais de detalhe, garantindo que a superfície base não se altera. Dessa forma, nosso método não precisa de uma etapa de reconstrução. Segundo, todas nossas operações de edição são feitas em uma projeção ortogonal. Além de ser mais intuitivo para o usuário, normais em um mapeamento ortogonal são de fato uma imagem RGBN. Isso nos permite definir operações de normal incluindo combinação, esboço de detalhes, filtragens, deformações e síntese de textura.

Além disso, nós apresentamos a arquitetura de um sistema interativo que implementa essas operações usando um atlas de textura para armazenar detalhes geométricos na forma de normais. Essa solução funciona com modelos em alta resolução e não impõe restrições sobre a topologia da superfície. Em particular, nosso sistema usa gráficos baseados em pontos para construir cartas ortogonais sob demanda. Em seguida, processa as normais e atualiza o atlas de textura. Por usar uma carta de edição temporária ortogonal, nosso sistema apresenta ao usuário uma representação transparente. Ele não se restringe a processar as cartas originais.

Palavras-chave: normal maps, imagem RGBN, detalhes

Abstract

Normals have been used in geometry acquisition and rendering for years. However, modeling with normals is rather limited because of the lack of formally defined operations. We present a variety of formal surface normal operations built on two key ingredients. First, a separation of models in a base surface and details, while the base surface is represented by positions in a mesh, the details are only represented through normals in a parametric space. Our operations can process the detail normals, while guaranteeing that the base surface remains unchanged. In this way, we do not require a reconstruction step. Second, all our editing operations are performed in an orthogonal projection. Besides being more intuitive for the user, normals in an orthogonal mapping are in fact an RGBN image. This lets us define normal operations including combination, sketching features, filtering, warping and texture synthesis.

Furthermore, we present an interactive system that implements these operations using a texture atlas for storing geometric detail as normals. This solution works with high resolution models and imposes no restrictions on surface topology. In particular, our system uses point-based graphics to build custom orthogonal charts. Next, it processes the normals and updates the texture atlas. By using a temporary orthogonal editing chart, our system presents the user with a transparent representation. It is not restricted to processing in the original charts.

Keywords: geometric modeling, normal maps, RGBN image, details, base surface, filtering, warping, texture synthesis, orthogonal chart.

"Corroborative detail will not produce a generalization every time, but it will often reveal a historical truth, besides keeping one grounded in historical reality. [...] That is the kind of detail which to me is worth a week of research. It illustrates the society, the people, the state of feeling at the time more vividly than anything I could write and in shorter space, too, [...] It epitomizes, it crystalizes, it visualizes. The reader can see it; moreover, it sticks in his mind; it is memorable."

Barbara W. Tuchman, Practicing History, p. 35

As for the author of this work, details were worth two years of research.

Contents

1 Introduction **1**

1.1 Representing Details 1

1.2 Normal Maps 5

1.3 Problem Statement 9

2 RGBN Image Processing **13**

2.1 Related Work 14

2.2 Filtering 15

2.3 Linear Combination 21

2.4 RGBN Image Warping 23

2.4.1 Creating Features 26

2.5 Nonlinear Normal Editing 30

2.6 Conclusion 33

3 Normal Synthesis **35**

3.1 Introduction 35

3.2 Related Work 36

3.3 Method 37

3.4 Jumpmaps on RGBNs 39

3.5 Normal Textures 42

3.6 Integrability Analysis 45

3.7 Results 47

3.8 Conclusion 48

4 Chart Editing **53**

4.1 Related Work . . . 53

4.2 Scale Decomposition . . . 57

4.3 Detail Processing . . . 59

4.4 View-Dependent Processing . . . 62

4.5 Atlas Representation . . . 63

4.6 Editing Chart . . . 66

4.7 Editing Normals . . . 73

4.7.1 Combining Normals . . . 74

4.7.2 Sketching Features . . . 79

4.7.3 Filtering . . . 79

4.7.4 Editing Multiple Charts . . . 83

4.8 Tangent Plane Method . . . 86

4.9 Conclusion . . . 89

5 Chart Transfer **91**

5.1 Related Work . . . 92

5.2 Method . . . 97

5.3 Chart Transfer . . . 99

5.3.1 Triangle-based Method . . . 99

5.3.2 Point-based Method . . . 101

5.3.3 Storage Update . . . 107

5.4 Chart Distortion . . . 108

5.5 Conclusion . . . 111

6 Conclusion **115**

6.1 Principal Contributions . . . 115

6.2 Discussion . . . 116

6.3 Future Work . . . 118

Chapter 1

Introduction

The quest for realism has always been the grail of computer graphics. Generating photo-real 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 representation and editing of fine surface details. More specifically, we process details represented by normal maps. We have developed normal operations both in the context of a single enhanced photograph and in complete surfaces. The operations we propose include filtering, texture synthesis and feature design through painting.

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, in terms of both 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 continuous multiscale representations, e.g. [1], it is easy to simplify complex models 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 most multiscale schemes is that they treat all scale levels equally, representing all of them as meshes. However, 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 not pointing towards the light. Its exact position is less relevant. Fine scale geometry (microfacets) 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 for rendering.

It is common to use three different scale levels [2]. The *macrostructure* level is the coarsest one. It is usually represented as a triangle mesh or a spline surface. 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. For example, the realism or credibility of the images in Figure 1.1 is provided by its mesostructure: dinosaur scales and human skin wrinkles.

There are many methods to represent the different structure levels. On the one hand, macrostructure is usually represented as a triangle mesh, NURBS, subdivision surface or an implicit surface. On the other hand, microstructure is usually represented by material properties either as a full BRDF or its approximation in a lower-dimensional function space as in the Phong shading model. In this work, we will not further discuss microstructure, instead we will focus on macro and mesostructure.

We will represent a *surface* by the combination of its macro and mesostructure. We shall use the words *base surface* to mean the macrostructure and *details* to refer to its mesostructure. For reasons that will be clarified shortly, we regard the details as a function defined in the base surface.

It remains an open question how to represent geometry mesostructure. The sim-

Figure 1.1: The realism or credibility of the images in this figure is provided by its mesostructure: dinosaur scales and human skin wrinkles.

plest way is to use a triangle mesh. If the triangles are small enough, mesostructure will be represented. However, since a mesh does not assume regular sampling, it must store the connectivity of its vertices and not just the 3D position of each vertex.

While mesh vertices are scattered in space, texture pixels are laid in a uniform grid. Therefore, textures maps have reduced storage requirements. For this reason, textures provide an adequate solution for representing attributes because they 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 not only many different color attributes like diffuse and specularity colors, but also geometric attributes.

Storing geometry in textures has further advantages over meshes. First, images have a natural multiresolution structure. We can filter and resample images building a pyramid. For example, it is easy to leverage the power of image pyramids for rendering with the mipmapping technique, supported by most GPUs. Second, 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 a texture available it will only be consulted on a per-pixel basis and the rendering pipeline will be faster.

The arguments above suggest that storing details in a texture is better than a full

mesh. However, what geometric details attribute should be stored? There are many possibilities. A first option would be storing 3D positions themselves per pixel, a technique called geometry images [3]. Second, in bump mapping [4], a height map is used. This means beyond the base geometry, texture stores a displacement in the normal direction. The normal of this local height map is calculated and added to the base normal, the normal of the base surface, to obtain the final normals 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. This technique is called *normal mapping*. One advantage is that no tangent plane needs to be calculated or stored.

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, leading to visual artifacts [4]. Displacement Mapping [5] is an extension of these techniques in which the local height map is used to change position itself, however it has less hardware support.

In this work, we chose to represent mesostructure in the form of *normal maps*. Normal mapping provides a good compromise of quality and generality, being supported in many hardware architectures. In the next section, we provide some details of acquisition and representation of normals.

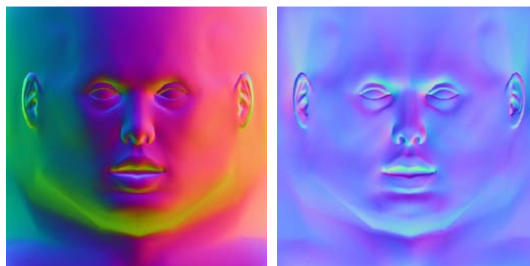


Figure 1.2: Object-space normal maps (left) have range in the unit sphere. Tangent-space maps (right) take values on a single hemisphere. Images taken from [6].

1.2 Normal Maps

Normal mapping stores the normals as a regular grid. During rendering, this map will be consulted and used to calculate shading. Since these normals will be stored at a higher resolution than the geometry itself, the model will look much more detailed.

The normals can be represented in two different ways [6]. In object-space normal maps (Figure 1.2) 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 in 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 wasting much of the resolution provided by 24-bit textures. In [6] other representations are discussed.

Having discussed representation of normal maps, we must discuss their acquisition. We can classify normal acquisition methods in two categories. First, there are methods that acquire or create normals directly, as discussed in the next paragraph. Second, there are methods that already start with a detailed model. For example, a high-res mesh (Figure 1.3-left) can be created with a modelling software or with 3D scanners. As we have argued, these very large models are unfit for interactive applications. Therefore, these meshes are frequently simplified (Figure 1.3-middle). To avoid the loss of quality due to simplification, normal maps are often created resulting in a surface appearence similar to the original when rendered (Figure 1.3-right). The authors of [7] use an appearence error metric to simplify a large mesh, storing

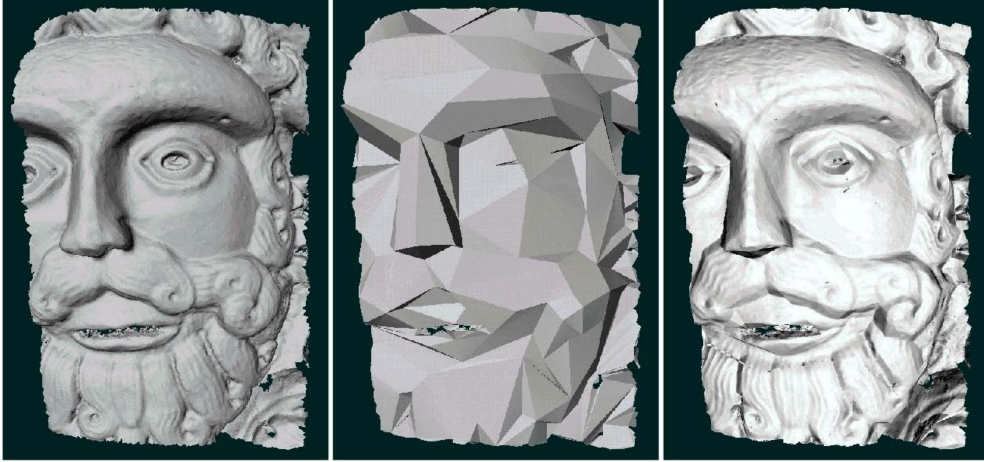


Figure 1.3: Normal maps can be used to enhance simplified meshes with little loss in quality. Images taken from [8].

color and normals in textures. This means pixels on a rendered image deviate little from the original color. In [8], attributes are transfered by sampling points on the faces of the simplified mesh and projecting them onto the high-res mesh.

However, starting with a high-res mesh is not easy. On the one hand, modelling is labor intensive. On the other hand, 3D scanners are expensive, single range scans have limited resolution and they have problems with high frequency noise. Acquiring normals directly is an alternative which can be accomplished with shape from shading (SfS) techniques [9]. These methods take a shaded image as input and aim at inverting the illumination equation to obtain shape. Since SfS works on common cameras, it allows very high resolution models to be obtained with low high-frequency noise. One important example of SfS methods is photometric stereo [10, 11] in which multiple photographs are used (Figure 1.4).

As detailed above, acquisition of normals is slightly more complex than usual photography, yet the power and flexibility of 3D models can be attained. For some applications, positions are not necessary, making the normal image returned from SfS an important data structure by itself. We shall use the term *RGBN image* to mean images that contain both color and normals per pixel. In Textureshop [12] the authors recovered normals from photographs and then developed texture synthesis

Figure 1.4: Photometric stereo uses multiple photographs are used. Images taken from [11] in which the authors acquired Michelangelo's Pieta.

on the resulting RGBN images. The normals are used to guide local distortions in the synthesized texture (Figure 1.5). In addition to realistic shading, many NPR rendering algorithms have been shown to work even if only a normal map is available [13]. 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 are easy to capture and easy to analyse.

However, shape from shading has its limitations. Typically, the normals produced have a low frequency bias. Nehab et al. [14] pointed out that a hybrid solution for geometry acquisition is necessary. They developed a method to integrate positions and normals. Their method unites the best of both worlds: the low frequencies from positions and the high frequencies from normals. The only drawback is that it results in a very high resolution mesh.

As described above, we follow a different solution in our work by separating the model in base surface and details. We only store the low frequencies of position as the base surface, thus having no problem with noise. On the other hand, normals represent details and are only used for shading. For instance, this mixed approach has been used to acquire Michelangelo's Pieta [11] (Figure 1.4).

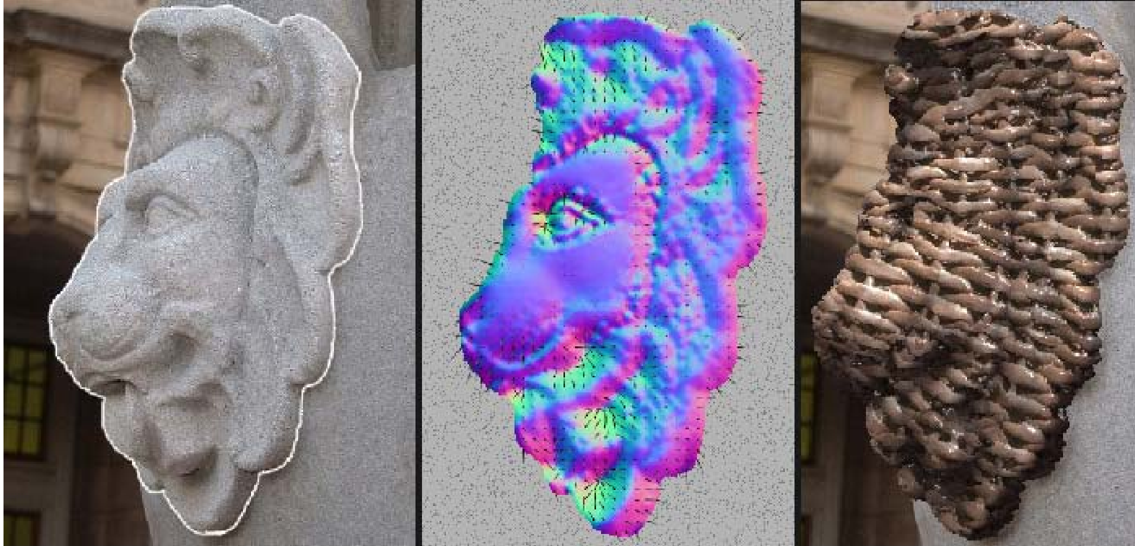


Figure 1.5: In Textureshop [12] the authors recovered normals from photographs and then developed texture synthesis on the resulting RGBN images.

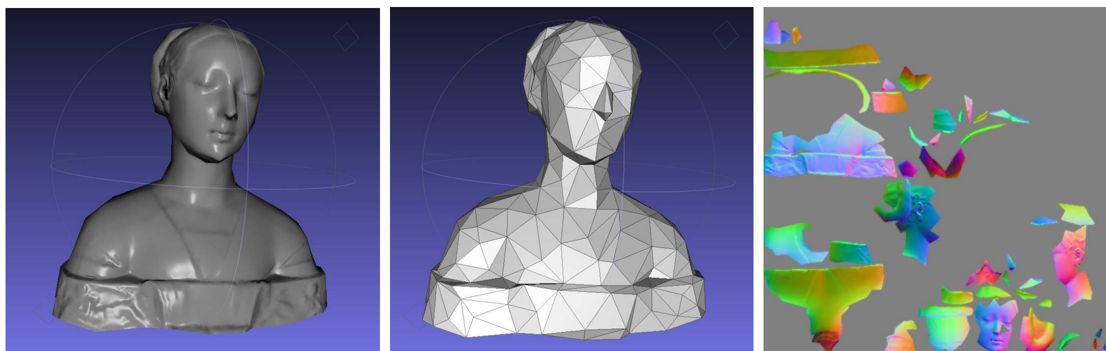


Figure 1.6: We use a texture atlas with two attribute functions defined: fine scale normals for details and coarse scale positions for the base surface.

1.3 Problem Statement

In the previous section, we have seen how normals are relevant in capturing and rendering fine details. We know how to capture normals and we know how to render normals, but *how should we process normals*? This is the main problem addressed in this work: given a fixed base surface with details represented in a normal map, how can we edit these details.

At this point, it is important to make clear the distinction between processing and creating details. Processing supposes there are existing details that will be changed somehow. On the other hand, creation starts from scratch. In this work, our goal is to *process* normals. These normals will usually come from captured data of real objects, but may also be applied as a post process to user-designed details.

We propose a full solution ranging from the mathematical definition of normal operations to its computational aspects and user interface. We present a solution with the following desirable characteristics:

- Ability to work in selective frequency bands, respecting the base surface/details separation;
- Ability to handle two-dimensional surfaces of complex topology;
- Intuitive and interactive editing;
- Workspace with realtime lighting feedback (WYSIWYG);
- Representation of very high resolution details;
- Transparent representation of data.

In a nutshell, we use a texture atlas (Figure 1.6) in order to map the surface to texture space. This way we can work with objects of complex topology by splitting it in pieces that can be mapped to planar regions. We require two attribute functions defined on this atlas: fine scale normals for details and coarse scale positions for the base surface. In addition, other attributes like diffuse albedo, specular albedo,

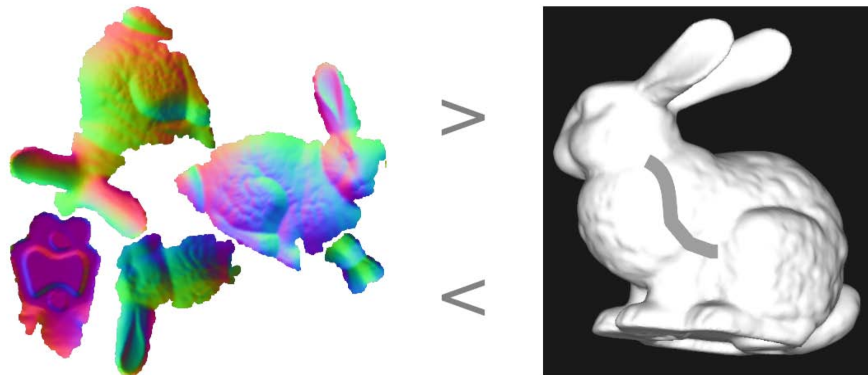


Figure 1.7: We project attributes, creating the temporary chart. All operations are performed in this chart. Finally, all changes are mapped back to the original atlas.

ambient occlusion or any other can be represented in the atlas and used for high-quality rendering.

Having settled on this representation, we will now provide an overview of how we approach normal editing in this work. A simple solution would be to let the user edit the charts directly. However there are problems with this solution. First, it is neither transparent nor intuitive since it requires direct access to the textures which include parametrization distortions. Second, some operations require calculations to be performed on an entire neighbourhood of a point. As can be seen in Figure 1.6, there are points in the interior of a chart that are very close to the chart's border.

For these reasons, we chose to perform all editing operations in a temporary chart custom built for each operation. In fact, this chart is just a projective mapping of the surface. As we will show the projective chart has many advantages. In particular, powerful normal operations are easier to define in this mapping.

So now, we have all the ingredients to state our editing process (Figure 1.7). We start with the attributes in the atlas. Next, we project them, creating the temporary chart. All operations are performed in this chart. Finally, all changes are mapped back to the original atlas.

During editing, we have color and normal attributes defined in the temporary chart. If we think of the attributes themselves and abstract the mapping process,

we are left with RGBN images. Editing these RGBN images will be the focus of the first two chapters of this work.

In Chapter 2, we present operations to edit the normals of the RGBN, including painting normals with brushes and filtering. First, this chapter presents two methods of combining details with the base surface and how this separation is important for normal operations. Second, it develops a framework for filtering normals that allows for smoothing, enhancing and even high-pass filters.

In Chapter 3, we show how texture synthesis methods can be used to transfer normals and automatically change large regions. This chapter builds upon the filtering methods from the previous one. Filtering is used for all base and details separation.

In Chapter 4, the core chapter of this thesis, we present in an abstract level our surface normal editing process and how the RGBN operations from the first chapters are integrated. We use a separation between base surface and details. In this setting, we can change the details but respect the base surface geometry.

In Chapter 5, we show the computational and implementation details of the process. Two solutions are presented to transfer the attributes between the temporary chart and the atlas. The first one is based in triangle rendering while the second one is based in point rendering. Both solutions have advantages and disadvantages regarding the resampling of the attributes.

Chapter 2

RGBN Image Processing

During our surface normal processing method, we have color and normal attributes mapped to the temporary chart. If we think of the attributes themselves and abstract the mapping, we are left with RGBN images, i.e., images that contain color and normals per pixel. Editing these RGBN images is the focus of present and next chapters. In this chapter, we present operations to edit the normals of the RGBN, including painting normals with brushes and filtering. First, it develops a framework for filtering normals that allows for smoothing, enhancing and even high-pass filters. Second, this chapter presents two methods of combining details with the base surface and how this separation is important for normal operations.

Even though our temporary chart includes low frequency information on positions, we chose to study operations that work solely on normals. While this brings some additional difficulties, it also makes the operations in this chapter more general. All operations apply to any RGBN image, in no way restricted by our surface normal editing process. Chapter 4 shows how these operations are integrated in the entire editing process. At the same time, it overcomes the main limitation of the RGBN image, having a fixed point of view.

This chapter is organized as follows. In Section 2.2, we propose a method for filtering normals allowing smoothing, edge enhancement and high pass filters. In Section 2.3 we introduce the linear combination method for adding details. Section 2.4 introduces RGBN image warping and defines the creation of features by

twisting some known detail along a user-defined path. In Section 2.5, we also propose a nonlinear operator to add details.

All the methods described in this and the next chapter were implemented in an RGBN image manipulation program as described in a technical report by the authors [15].

2.1 Related Work

Toler-Franklin et al. [13] coined the term RGBN to refer to an array of pixels with associated color and normal channels. They have shown that many NPR rendering algorithms work in RGBNs. These include toon shading, line drawing methods, curvature shading and exaggerated shading. They developed signal processing techniques like low-pass filtering, derivatives and curvature estimation.

ZBrush [16] is a digital sculpting tool. It can create high-resolution models using subdivision. They can then be painted/sculpted using 2.5D images of *pixols* which contain colors and depth. Normal maps can be generated from the final models. For editing normals, they would first have to be converted to depth information outside ZBrush.

In [17, 12] the authors recover normals from photographs and then develop texture synthesis on the resulting RGBNs. This way the normals are used to guide local distortions in the synthesized texture. In Textureshop [12], the authors also transferred normals between images. Poisson image editing [18] was used to merge normals seamlessly, followed by normalization, a process which does not guarantee a conservative normal field.

Normalpaint [19] proposes a tool for creating normal maps directly, thus avoiding the need to model a high-res mesh in the traditional modeling pipeline. They calculate normals in a way that is equivalent to height extrusion. In their approach only shapes with simple geometry can be created and editing of normals is not supported.

In Gradient Domain Painting [20], the authors propose tools that change the

gradient of color images, a GPU-multigrid integrator recovers the new image in real-time. Different gradient blending modes are shown. In our work, the entire pipeline is composed of local operations.

In [21], Taubin defines a specialized laplacian operator for filtering normals on a mesh. He then integrates this normal field to obtain filtered positions. Different smoothing filter for normal maps have been developed for mipmapping. In [22], the author smooths normals and shows how the shortening they introduce can be used to reduce aliasing of specular highlights. In [23], the authors formalize normal map filtering using convolution between normal distribution functions and the BRDF.

2.2 Filtering

The simple way to filter an RGBN is 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. As Toler-Franklin et al. [13] pointed out, the problem with naive filtering is that due to foreshortening the area of each pixel will be underestimated by $\cos\theta$, where θ is the angle between normal and viewing directions. To simplify the analysis the authors 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_1, n_2, n_3) by $(n_1/n_3, n_2/n_3, 1)$, which we call *foreshorten corrected*. In this representation, filtering is now a linear operation as long as the third component is preserved. We next show that we can actually ignore the third component, allowing us to use any linear filter.

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 $z_x(x, y), z_y(x, y)$. Using the surface parametrization

$\psi(x,y) = (x,y,z(x,y))$ whose tangent vectors are $\psi_x(x,y) = (1,0,z_x), \psi_y(x,y) = (0,1,z_y)$. We obtain the normal vector:

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

The above formula lets us convert from z_x, z_y to N. In fact the foreshorten correction scheme shown above is the reverse process. Given a unit normal $N = (n_1, n_2, n_3)$:

$$-n_1/n_3 = z_x, -n_2/n_3 = z_y$$

This can be done as long as $n_3 \neq 0$, otherwise there is no height map that represents this surface. What we have shown is a one-to-one mapping between the N and z_x, z_y, as such, we can work with one or the other indiscriminately. Note that we use the terminology z_x, z_y loosely here, since this field might not be the gradient of any C^2 height function. Conditions for this to be a gradient will be a recurring theme in this work.

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}. 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}, as shown below:

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

We want to avoid surface reconstruction. Fortunately, the arrow (1) above will provide a shortcut since derivatives and convolution satisfy the relation $(z * g)_x = z_x * g$.

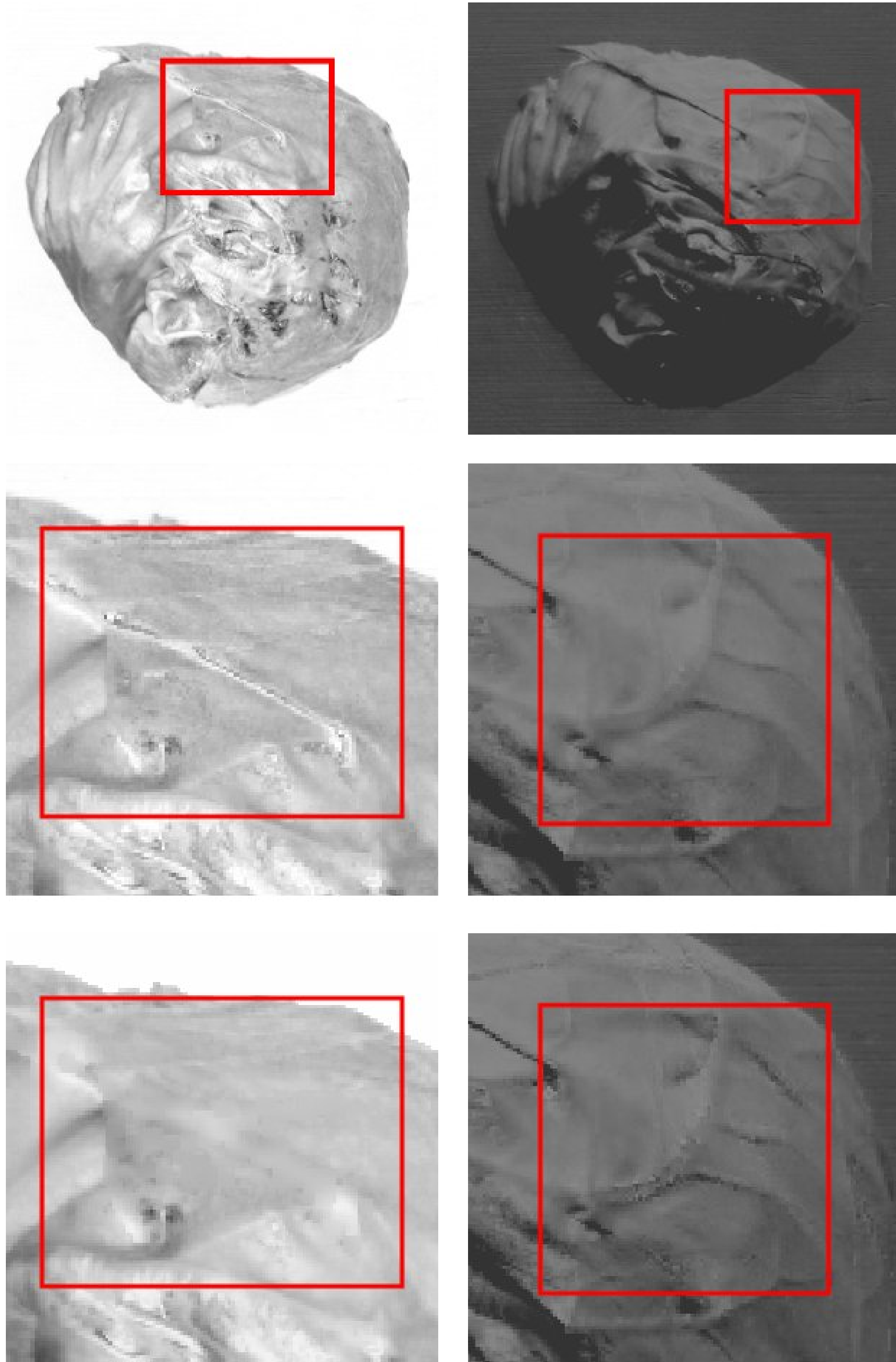
This means when the normal is foreshorten corrected $(-z_x, -z_y, 1)$ we can convolve it with any kernel. Since we are only interested in filtering z_x, z_y, we can simply set the third component to 1, whether the kernel would preserve it or not. After

filtering we simply normalize the vector to get a unit normal back. Notice that the resulting field is guaranteed to be conservative, since it is by definition the normals of a height function.

Having developed this filtering framework, we now discuss some applications. Gaussian Filtering, in fact, any low-pass filtering would smooth the normals (Figure 2.2). We are also interested in Sharpen Filters to enhance detail, i.e., an all-pass filter + high-pass filter. The problem with sharpening is that by enhancing high-frequencies we also enhance noise (Figure 2.1). A simple solution is using an Edge Enhancement Filter using all-pass + band-pass.

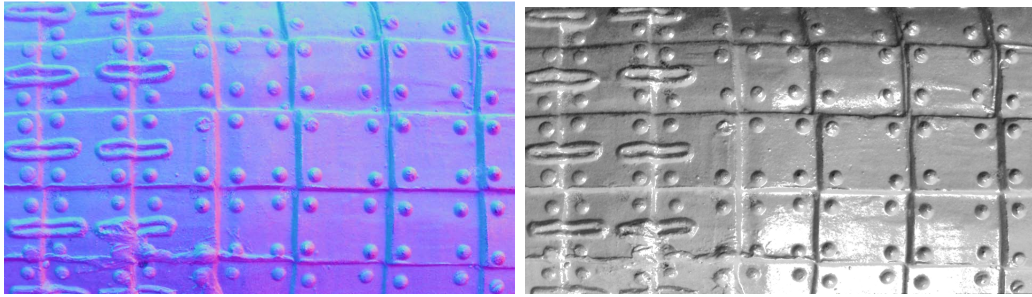
All of the above filters preserve the DC frequency. Filters that do not have this property can be useful for editing normals. For example, we might be interested in extracting a normal texture from an RGBN. In this case, we look for eliminating the low-frequencies related to shape and retaining the high-frequencies related to texture (Figure 2.3). A Difference of Gaussians and a Laplacian of Gaussian provide simple band-pass filters giving us the texture but also eliminating high-frequency components like noise. Another application is simply scaling the surface represented by the normals generating shallow surfaces.

We have developed a local filtering operator by applying the above filtering procedure in a small neighborhood. The user defines the shape and radius of the region. In Figure 2.1, we show smoothed and enhanced normals. Noise was also enhanced in this example. The local filter works well in practice, but it can be regarded as a filter with a spatially varying kernel, since it leaves most of the image unaffected. For this reason, the above integrability property may not hold.

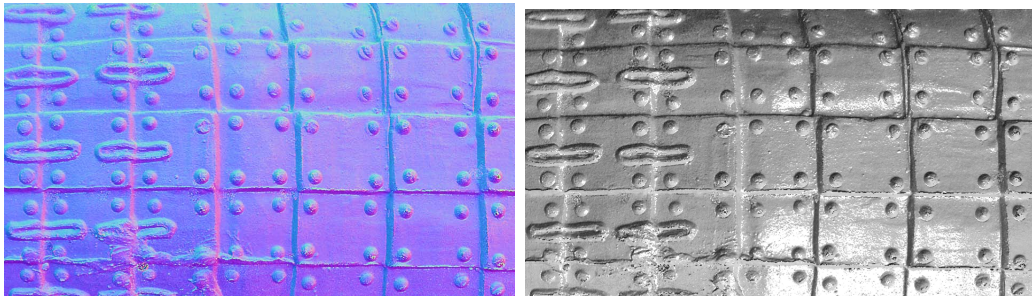


(a) Smoothed Normals (b) Enhanced Normals

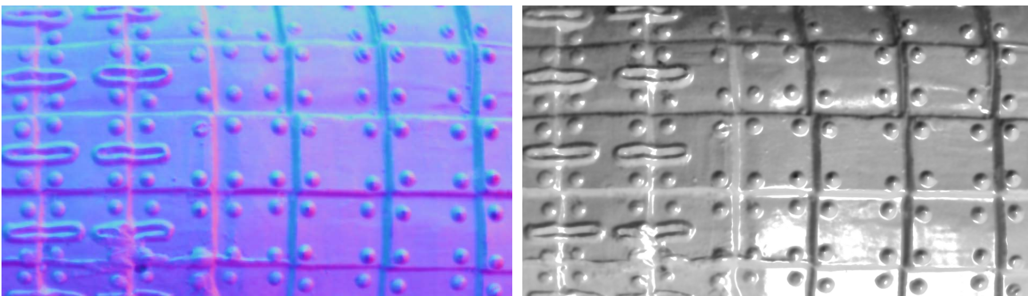
Figure 2.1: Normal filtering can be applied locally as a brush.



(a) Original Normals

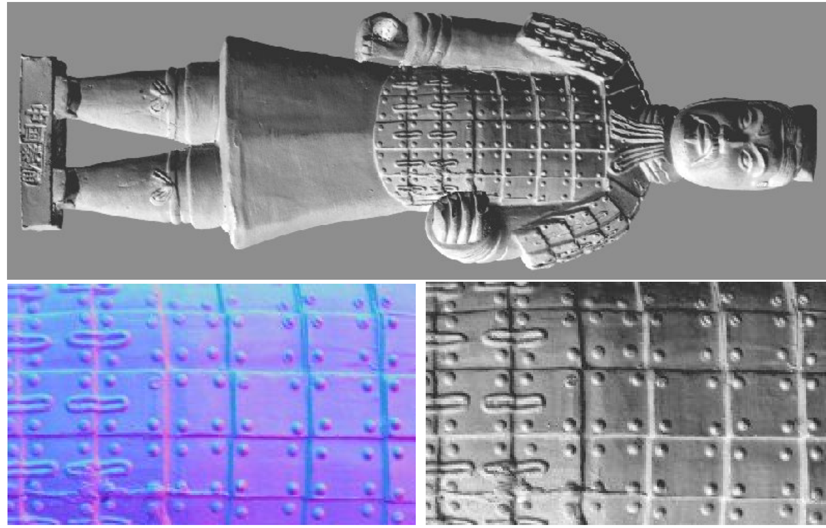


(b) Enhanced Normals

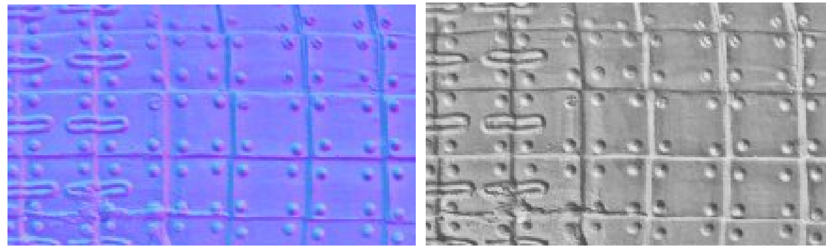


(c) Blurred Normals

Figure 2.2: With our framework, any linear filter can be applied to normals.



(a) Original Normals



(b) High-pass filter result

Figure 2.3: A high-pass filter was used to remove the shape and retain a flat normal texture. All shaded images in this work are generated with directional lights.

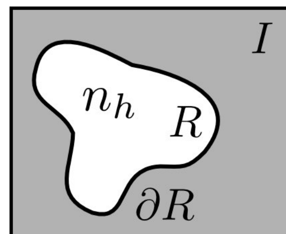


Figure 2.4: The detail normals n_h defined in R are combined with the image normals.

Figure 2.5: 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.

2.3 Linear Combination

In this section, we investigate a method of adding details to RGBNs. Details could be an applied stamp (Figure 2.5) or bumps painted by the user (Figure 2.11).

Given a base normal field n_b defined in the entire image I (Figure 2.4) and a detail normal field n_h defined in $R \subset I$, the problem of combining normals is generating a new normal field w which agrees with n_b everywhere, but is influenced in R by n_h. We refer to their respective height functions as b and h. We would like combination to be:

1. integrability preserving: lead to normal fields that correspond to a real surface;
2. frequency preserving: respect or replace selective bands.

Notice that simply adding the normal vectors and renormalizing does not satisfy any of the above properties. We propose the linear combination model which is integrability and frequency preserving. This is crucial for editing normal maps, since we usually want to edit mesostructure (normals) without affecting macrostructure possibly encoded in a different representation.

In the linear model we would rather look at the normal fields as derivatives. Just like in the previous section, suppose we could add height functions b and h, respectively base and details. By linearity of the gradient, $w = \nabla(b+h) = \nabla b + \nabla h$. So even if we do not have heights, we can still obtain the new gradient and thus the new normals. The new field is trivially conservative. It is also frequency preserving, if a frequency band is not present in the details, it will be unharmed in w. To replace a given band, we can use a band removal filter in the original RGBN either globally or only in R, thus building the n_b normals which can then be transformed in the ∇b used above. This way we can respect not only low shape frequencies but also high texture frequencies, only changing mesostructure. Section 2.5 also presents a nonlinear combination.

Now that we have a good understanding of combination, we can look at how to specify the details. The first local operator we propose is inserting a small RGBN (stamp) in a neighborhood of a point (Figure 2.5). The stamp can be an entire RGBN itself or extracted from such. In the last case, we may want to eliminate the lower frequencies through filtering when creating n_h. Notice that if we want to apply any 2D linear transformation A to the stamp, gradients will not be simply copied like colors, instead we should transform them by $(A^{-1})^T$ since they are contravariant. This allows us to scale, rotate or shear. Translations do not affect normals. In the next section, we propose RGBN image warping which generalizes all these transformations. In addition to the stamp operator, we present a method to create new details.

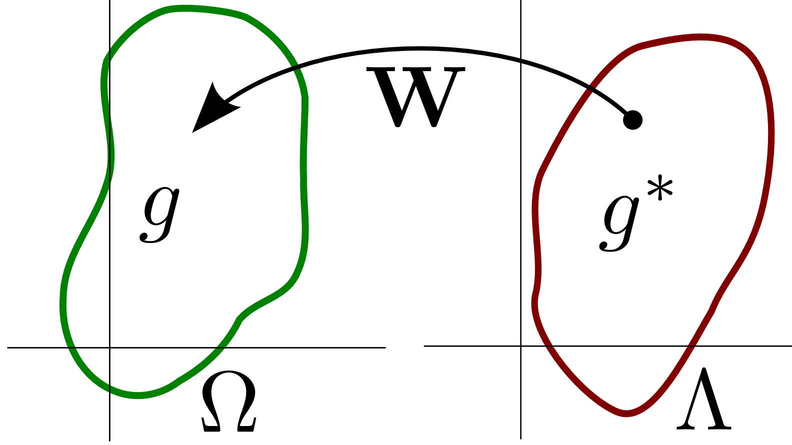


Figure 2.6: The domains Ω and Λ, and its gradients fields g and g^*

2.4 RGBN Image Warping

In the image warping problem, we are interested in transporting attributes from one domain Ω to another Λ. We assume both domains are related by a diffeomorphism $\mathbf{W}$. Transporting color amounts to using the field $\mathbf{W}$ to evaluate the color function in a reference point in Ω. This simple solution fails for warping normals. Normals have to rotate, twist and stretch according to the warping field. Our solution consists of looking at normal vectors as height gradients and warping gradients. We do not want to obtain a height map explicitly, but it is a good abstraction to warp normals. The conversion between gradients and normals was already presented in this chapter. We next present a gradient warping method.

Given two domains $\Omega, \Lambda \subset \mathbb{R}^2$ related by a warping diffeomorphism $\mathbf{W} : \Lambda \to \Omega$ (Figure 2.6). Given a gradient field $g : \Omega \to \mathbb{R}^2$, define $g^* : \Lambda \to \mathbb{R}^2$, $g^*(x) = g \circ \mathbf{W}(x) \cdot D\mathbf{W}(x)$. We say g^* is the field g warped by $\mathbf{W}$. In other words, apply the chain rule. If we look at $\mathbf{W}$ as $(u, v) = \mathbf{W}(x, y)$, we may write in matrix notation:

$$g^* = g \cdot D\mathbf{W} = \begin{pmatrix} \frac{\partial z}{\partial u} & \frac{\partial z}{\partial v} \end{pmatrix} \begin{pmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} \end{pmatrix}$$

This gradient warping rule is equivalent to copying the heights directly. The difference is we do not have heights, only gradients. To show the equivalence we use the chain rule, $g^* = g \cdot D\mathbf{W} = \nabla z \cdot D\mathbf{W} = \nabla(z \circ \mathbf{W})$. Therefore, g^* is the gradient

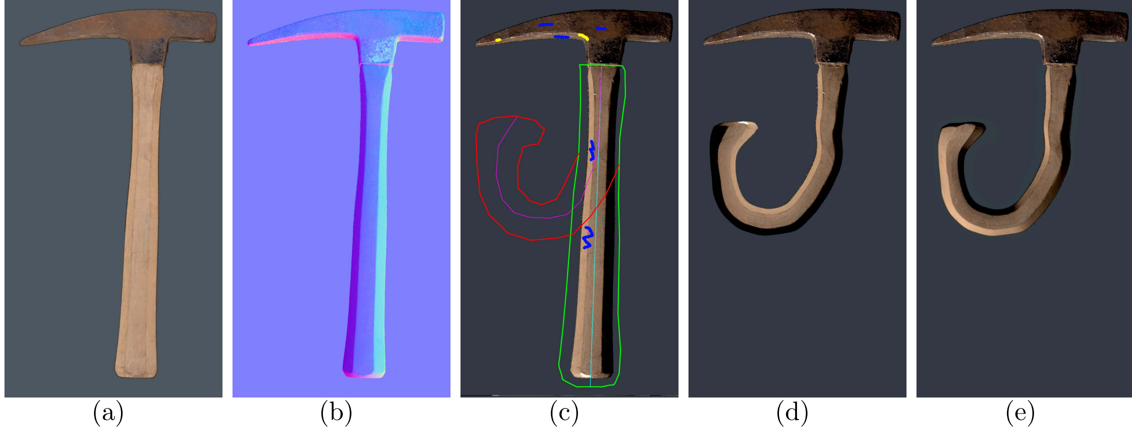

(a) (b) (c) (d) (e)

Figure 2.7: Original RGBN image (a,b). User inputs two groups of sketched strokes (c) (green and cyan for border sketches, red and purple for deformation interior sketches); (d) final result with simple color warping; (e) final result using warped normals for relighting.

of the warped height function. In short, to warp normals we first convert them to gradients, warp the gradients and finally convert them back to normals. Next, we propose two applications of our warping framework. First, we show results of using multiple sketches to define an arbitrary warping of an RGBN image. Second, in Subsection 2.4.1, we use single sketches allowing creation of new details such as new features.

In the next results, the warping fields were generated using the sketch-based method presented in [24]. Two kinds of sketches are used: boundary and internal sketches. While boundary sketches relate the borders of Ω and Λ, interior sketches specify the mapping in the interior of these regions. As can be seen in Figures 2.4(d) and 2.4(e), RGBN image warping can be used with relighting to produce correct shadows and highlights. The same comparison can be done in Figure 2.8 or in Figure 2.9, where we can see the new volume through highlights and shadows.

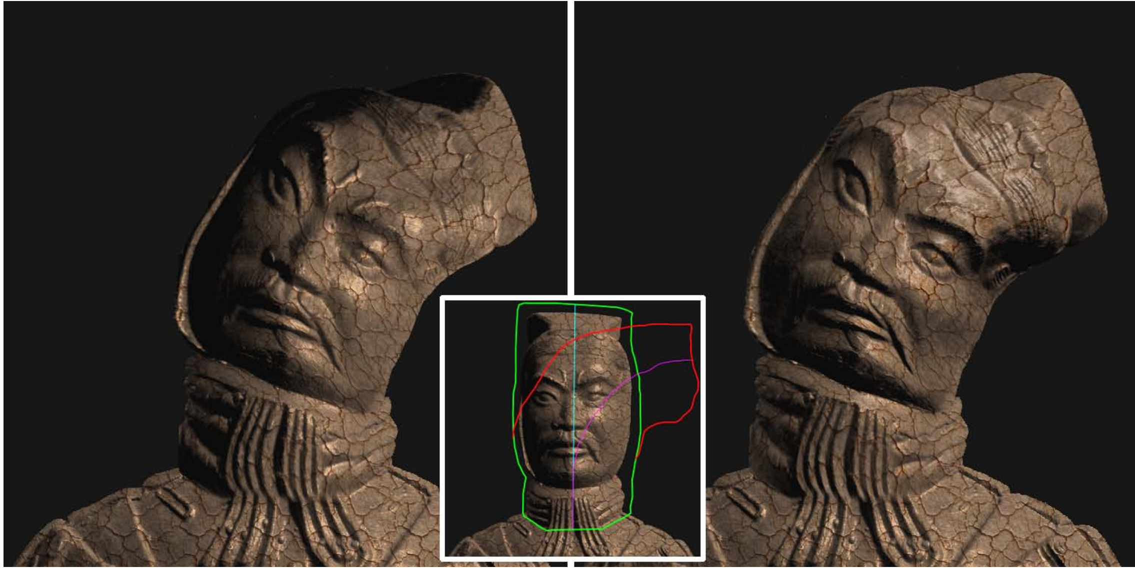


Figure 2.8: Results of deformation using our system in the Soldier RGBN image: upper left the result without relighting, upper right the result with relighting with the correct warped normals.

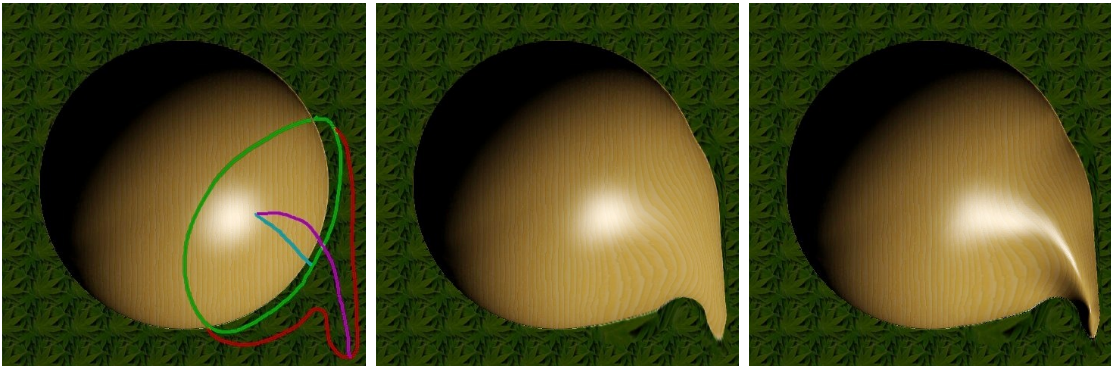


Figure 2.9: First, warping sketches define the free-form deformation. Second, we present the final result without relighting and the final result relit with correct normals.

2.4.1 Creating Features

To create line features we use the pen operator. In Figures 2.10 and 2.11, custom profiles are used to create bumps, creases or scratches on the surface [1]. A deformation $h(u,v)$ defined in a canonical domain has to be warped along an input path. The first step is to establish the warping mapping ϕ in a tubular neighborhood of the path (Figure 2.12) drawn by the user. The $u(x,y)$ coordinate can be regarded as a radial displacement from the path (distance), while the $v(x,y)$ coordinate as a displacement along the path (ideally arc-length parameterized). In the pen operator the profile is independent of the v coordinate, which simplifies the calculations. In case we do not have $h(u,v)$, only its derivatives, we use the gradient warping method define above.

To avoid slow distance field calculations, we use geometrical methods using point to segment distance functions and projections to build $\phi(x,y)$. This might not be the exact distance field because we calculate it locally and online as the user draws. 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? We analyze this question in the next section.

We have shown above how linear combination results in conservative fields. In addition, we have also seen how with a proper gradient warping rule we can produce warped conservative fields. However, we must make some additional considerations concerning the pen operator. First regarding the differentiability of the mapping W in the curve neighbourhood. Second regarding the continuity of the resulting surface, specially in the border of the deformed neighbourhood.

Let's assume the curve C defining the edited region is a C^2 regular curve ($C'(t) \neq 0$). We can always approximate discontinuities with high curvatures. Under this hypothesis, there is a tubular neighborhood T where the mapping defined by ϕ is a diffeomorphism [25]. In fact this holds if T does not intersect C's medial axis M

[1] Matching profiles and deformations is left as an exercise for the reader.

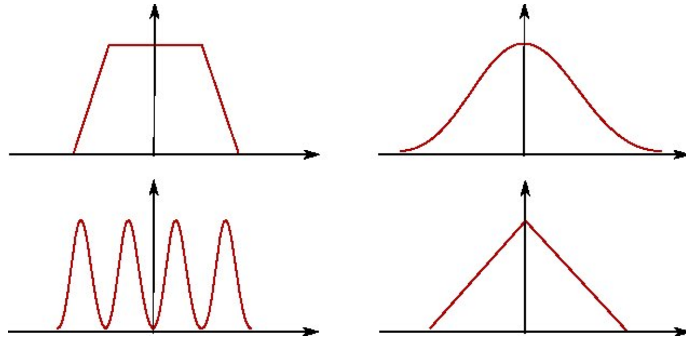


Figure 2.10: Custom height profiles along a path are transfered to the normals with the pen operator.

Figure 2.11: This cucumber was edited using the four profiles above. Height profiles are an intuitive way of specifying deformations.

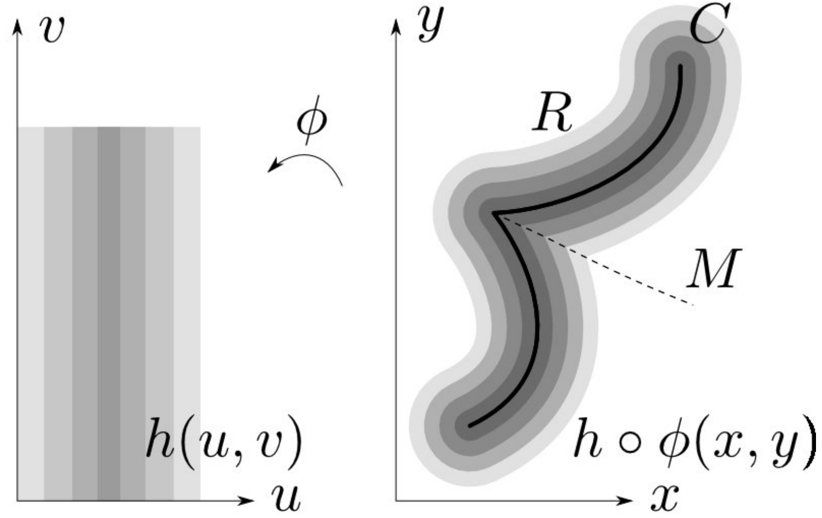


Figure 2.12: The distance field is used to build a mapping around the curve. This mapping is singular only over the path's medial axis.

(Figure 2.12). We have defined the curved profile normals as $w = \nabla h \cdot D\phi = \nabla(h \circ \phi)$, so w is conservative and C^1, assuming h is C^2. Note that we do not have $h \circ \phi$, but we know this function exists. Nowhere in the above demonstration, we used that $h(u, v)$ depends only on u (as is the case for profile editing), so our method works for more patterns like normal textures.

The problem with the above hypothesis is that M can be arbitrarily close to the curve, making R a very thin region. If this does not hold, we can still show equivalency with a continuous surface. If we require $h(u, v)$ to depend only on the distance parameter u, we can show that w is C^1 everywhere, except for the points of M, where w is not defined. In fact, there is a C^0 function h_2 defined in the image I such that w is the gradient of h_2 in $I \backslash M$ where $h_2 = h \circ \phi$. ϕ is C^1 outside M. h_2 is continuous on M because h depends only on $u(x, y)$ which is a continuous function even on M. This discontinuous derivative will lead to discontinuous normals (creases) in the final continuous surface. This characteristic might be desirable or not. If $h(u, v)$ depends also on v, w is the gradient of a height function which may be discontinuous in $I \backslash M$.

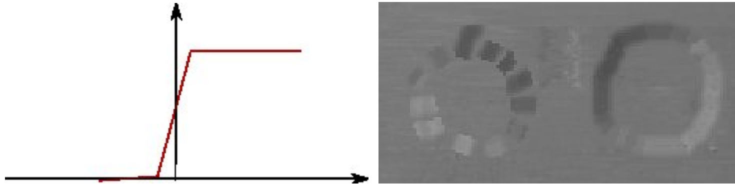


Figure 2.13: Notice how the shading of an impossible object (left) resembles a rotated version of a bump (right).

Both the stamp and the pen operator are only defined locally, this is equivalent to defining h as being zero outside the edited region R. This requires ∇h to fall smoothly to zero close to ∂R and be equal to 0 outside. For an open curve C, this is not enough. Figure 2.13 shows a counter-example based on Escher's infinite stairs. The paths are radial and the height profile was a step function as shown in Figure 2.13. Each edit is a step up, so we could climb this stair infinitely up. To fix this issue, we ask one more property of w:

$$a, b \notin R \Rightarrow \int_a^b w \, ds = 0$$

This means open curves cannot introduce level changes outside R. For closed curves with no self-intersections the integral restriction above can be made weaker, the integral between the left side (inside of the curve) and the right side (outside) needs only to be a constant, not necessarily 0. Editing with this kind of profile raises (or shallows) the region inside the curve (Figure 2.14), but w is still guaranteed to be conservative. Notice that in this case $h(u, v)$ should be defined in a domain with the topology of a cylinder. By invariance of line integrals through diffeomorphisms, these properties can be checked in $h(u, v)$. Further care needs to be taken near the medial axis. For the local filtering operator, the integral restriction does not necessarily hold, as such the resulting field may not be conservative.

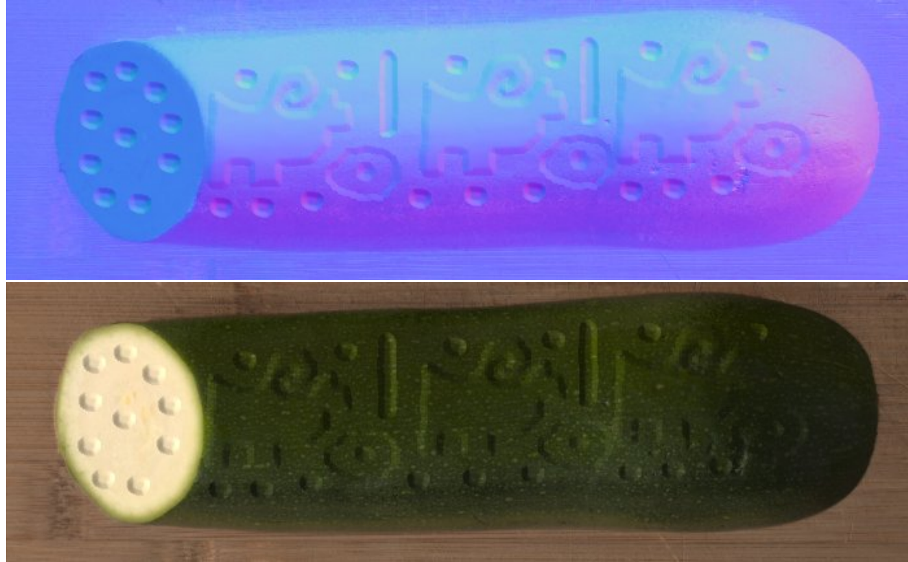


Figure 2.14: All carving was done with the pen operator. We can raise (lower) a region by tracing its border with a pen operator that introduces a level change.

2.5 Nonlinear Normal Editing

In the linear combination method, deformations are biased towards the camera (Figure 2.15). This problem is most noticeable near object silhouettes (Figure 2.17), since in these regions the normal is farther from the camera direction. This same problem is found in mesh deformation methods. We would like a method that wraps the deformations in the direction of the smooth normal. In this section we propose a nonlinear combination method that works in the local tangent plane, thus eliminating camera bias.

Deformations in the rotation method are in the base normal direction, but since it is a nonlinear method we need a different way to control integrability. We propose a two-band separation scheme. The signal is split in smooth frequencies and details. Our operator will only affect the details and respect the smooth frequencies. For this reason, the rotation model can only be applied to small scale features. This approach leads to good visual results. The extension of this framework for editing normal maps on arbitrary meshes should follow naturally by assuming the base normals to be the fixed mesh normals.

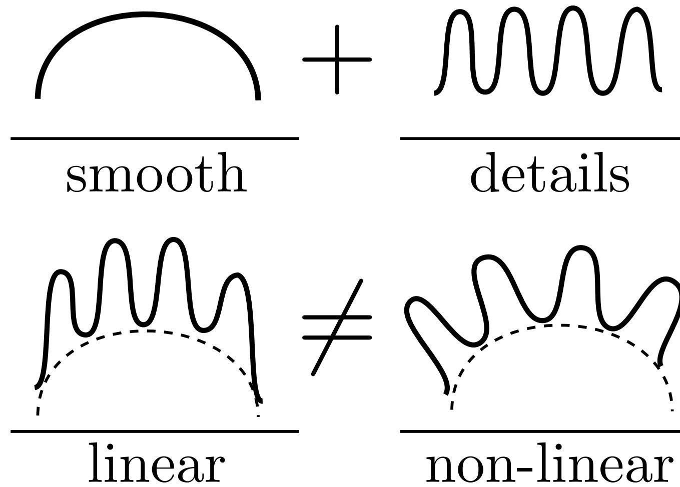


Figure 2.15: Comparison of the linear and nonlinear combination method.

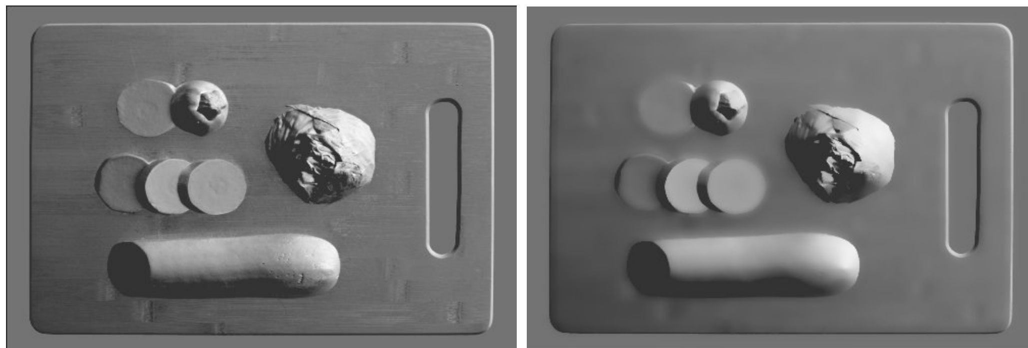


Figure 2.16: The original normal image is smoothed with bilateral filtering producing the base normals.

As in Figure 2.4, n_b denotes the base normal field and n_h a detail normal field defined in $R \subset I$. We want n_b to be smooth. The problem with building n_b with linear filters is that edges are not preserved. Even though pixels near an edge are very close spatially their colors (or normals in our case) are not correlated. The idea of bilateral filtering [26] is to take into account not only domain weights (distance) but also range weights (color similarity). In [13], bilateral filtering was extended to RGBNs, using also normal similarity (see Figure 2.16). This filter produces great results visually, but it is not know if it is an integrability preserving operation.

In the rotation model, we define a local coordinate system in each point (tangent

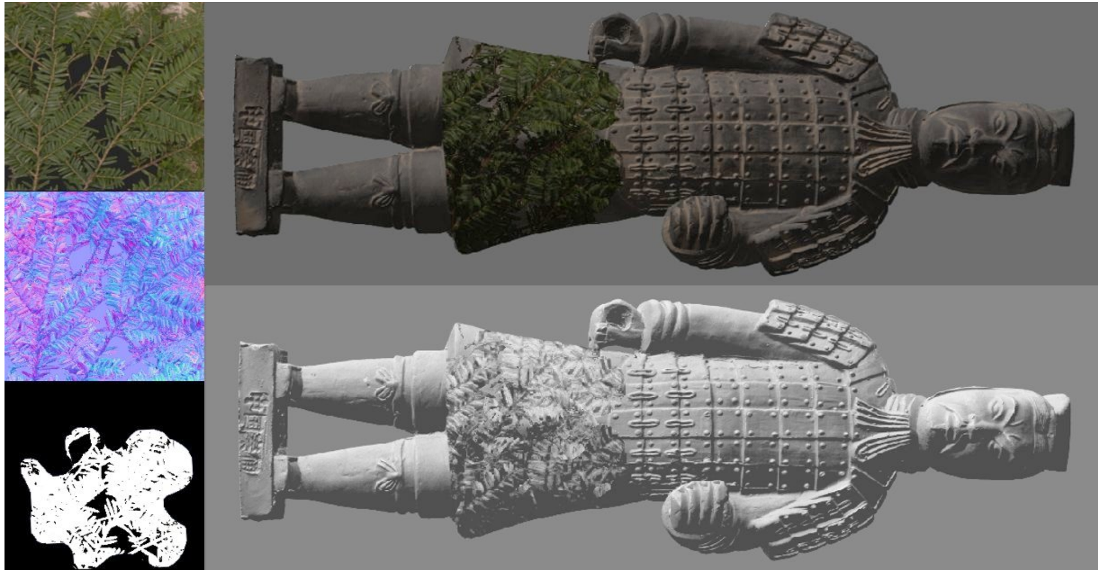


Figure 2.17: Branches were created on the soldier's skirt. On the left, the RGBN with the branches and a binary mask used to diffuse the regular square borders. The stamp was applied 3 times and segmentation methods were used to restrict editing to the skirt.

space) using n_b. We interpret n_h as being in this local system. We compose the two normals with rotations. Our local operations are transformations in a local tangent coordinate system u, v, n defined by the base normal vector n and an orientation vector o in the xy plane. Using rotations, it is easy to work on the tangent plane as well as interpolate operations. Editing the normal starts by specifying a desired normal vector b. We can simply replace the existing normal a with b. Another option is to blend the two vectors. Blending can be very useful for fading smoothly the effect of the new normals as distance from the edited area increases. We can also preserve frequencies if we rotate the existing normal a (including high frequencies) by the direction and angle specified by b (medium frequencies) in the local tangent system n_b (low frequencies). This strategy fails when b (in global coordinates) has a negative z value. 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 larger than a threshold.

2.6 Conclusion

RGBN images are one of the simplest ways to enhance color images with geometric information. They are very easy to capture, requiring inexpensive hardware and achieving better quality in high-frequency information than 3D scans. In the future, a great number of users should be able to work with RGBN images and achieve some effects that were only possible with the use traditional modelling tools.

We have shown a method specifically designed to edit RGBN images. Our system can filter normals allowing for low-pass, high-pass and edge enhancement. It contains brushes for adding detail and creating new features giving the user great control. In addition, we present methods to warp RGBN images. Conditions were established that guarantee integrable results.

However, our method has limitations. First, requiring the deformations to extend to 0 outside R is a limitation of our method, blending techniques should be investigated in the future to enforce this property on any deformation. Since the Jacobian $D\phi$ is used to warp the normals, the method would profit from high quality mappings around the curve. This is a problem only for deformations that depend on the arc-length parameter in high curvature regions.

If real-time feedback is not required, an exact distance transform can be calculated. This would allow curves of topology different from the unit real interval, like a T or H-shaped curve which are currently not supported by our local distance calculation. For profile editing, even curves with self-intersections would be possible. A different approach with the current system is to use the closed line pen operator to trace the border of a thin region, approximating the curve.

In addition, it is possible but user-time consuming to simulate normal texture synthesis with the stamp operator (Figure 2.17). In the next chapter, we extend the work of Fang et al. [27, 12] to synthesize normals. It allows more advanced editing of normals and color in large regions.

The main limitation of the RGBN image is the impossibility of changing the view point. While in this chapter we see the RGBN image as the representation itself,

in Chapter 4 we use a normal mapped mesh to construct RGBN images on demand from any given point. Therefore, extending the operations developed in this chapter to manipulate normals in any surface.

While many of the possibilities opened by the method presented are explored in the next chapters, there is additional future work. As discussed, Toler-Franklin et al. [13] have developed many rendering algorithms for RGBN. In this work we focused on editing geometry, but more visualization methods could be designed, building on our general filtering method. Another line of work, focusing on appearance, is to work on the relation between reflectance properties, illumination conditions and geometry. We could paint highlights and find the appropriate light positions, or fix lighting and indirectly edit reflectance all on top of an RGBN.

Chapter 3

Normal Synthesis

3.1 Introduction

In this chapter, we use texture synthesis to edit large regions of an RGBN image changing both color and normals. Editing regions by individually addressing local changes can be painstaking to artists. For this reason this process needs to be made automatic. Two options arise. Procedural methods are very powerful but are in general difficult to control. We use a texture from example method that takes as input only a small sample of the desired texture and then reproduces it in a large region. Our system can synthesize normal textures on shapes represented by normals (RGBN images), never using positions. Regarding synthesis, an advantage of working with RGBN images is that the projective mapping distortions on the synthesized textures can be corrected by using information from the normals. The texture sticks to the surface. However, we do not want the texture to follow every small surface irregularity. Therefore, we separate the RGBN image into macro and mesostructure bands.

Capturing a model's normals is easy using photometric stereo techniques [10]. It takes as input multiple images from the same view point, each illuminated with different known light positions. It then solves a least-squares problem to find the normal and albedo (color) of each pixel in the image. Since photometric stereo works on regular images, it allows the capture of very high resolution models. This is a

powerful method to obtain full models but also to generate texture exemplars.

The main contributions of this chapter are:

1. Synthesizing normal vectors allowing relighting of the final model.

2. An RGBN image editing system implementing a frequency aware texture synthesis framework separating exemplars and models into base shape and details.

In Section 3.3, we present an overview of our normal synthesis framework. In Section 3.4, we discuss the work of [17] in color synthesis on RGBN images. In Section 3.5, we extend their method for synthesizing normals. In Section 3.6, we determine when the edited RGBN images correspond to a realizable surface.

3.2 Related Work

Example-based texture synthesis can be classified into two approaches. In pixel-based methods, new pixels are generated one at a time. In [28], the authors search for a best match between the neighborhood of the already synthesized texture and neighborhoods in the texture sample. There are also patch-based methods [29] [12] 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.

Another classification of synthesis regards the domain. Texture synthesis can be parametric, volume-based or manifold-based. Volume-based methods will synthesize texture usually in 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 [30] [31]. These methods work directly in the manifold representation (usually a mesh) and use only local decisions.

In [17, 12] the authors extend manifold techniques to work on RGBN images, using the normal as a local descriptor of shape. Both works use shape from shading to recover normals from photographs. Normals are used to guide distortions in the synthesized color texture. Zelinka [17] adapted jump map based synthesis which

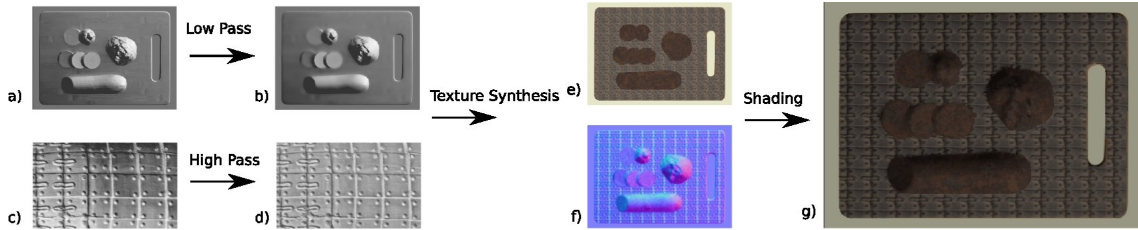


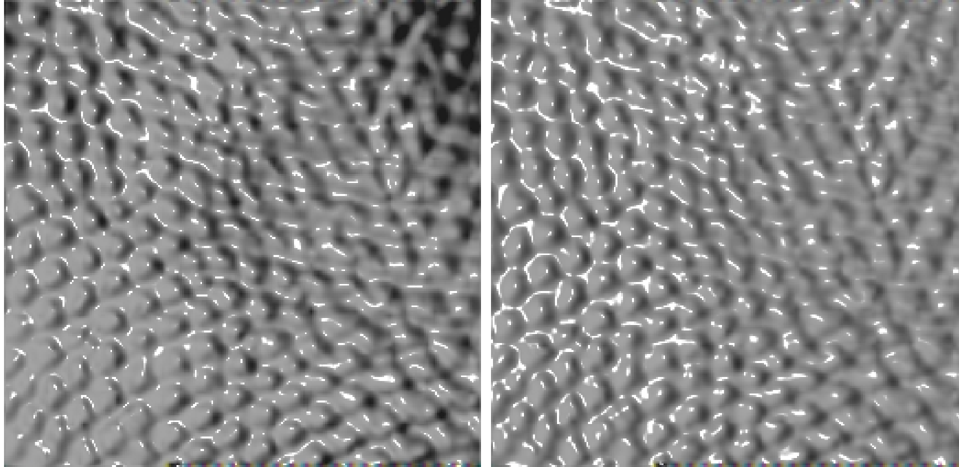
Figure 3.1: Overview of the method. Inputs: a) RGBN image, c) texture exemplar. Filtered inputs: b) smooth RGBN image, d) texture detail. Results: e) color, f) normal (represented in RGB colors) and g) shaded RGBN image.

is a pixel based method to work on normals, while Textureshop [12] uses a patch based method. Both works allow for advanced editing of images. For example, object material can be replaced, while still respecting shape and shading. Unlike these works, in our method, both color and normals are synthesized on RGBN images. One way of looking at RGBN synthesis is as a quasi-stationary process [32]. Synthesis proceeds through the image but varies spatially depending on the normals.

In [33], the authors capture high resolution normals of small patches of face skin and then use texture synthesis to replicate these tangent space normals in parametric space. Adding detail to faces is an important application of our method. While in their approach, a normal map is synthesized from scratch, we can respect existing frequency bands in the normal map. Procedural synthesis was explored in [34] where bump maps are created on the fly based on a normal density function.

3.3 Method

While the problem of synthesizing color textures on top of RGBN models has been studied in [17], we propose a method to synthesize normal textures. We start this section with an overview of our method (Figure 3.1). The method receives as input an RGBN image (a) and an RGBN examplar (c). It has three steps. First, in the frequency splitting step, we low-pass filter the normal image (b) removing details



(a) Original Exemplar (b) Filtered Exemplar

Figure 3.2: Removing the normal low frequencies of the exemplar only the desired details. We show shaded images of the normals under same lighting conditions.

and we high-pass filter the examplar normals (d) removing the base shape. Second, in the texture synthesis step, we synthesize both the colors (e) and normals (f) of the examplar on the smooth normals (b) compensating for foreshortening distortions. Finally, in the combination step, we merge the synthesized normal details with the smooth RGBN image.

To build the smooth normals we use the RGBN bilateral filter [13]. It takes into account both normal and color differences and respects edges. During splitting, we use the filtering method of Chapter 2 to apply a high pass kernel on the texture sample (Figure 3.2). As an alternative to the bilateral filter we could build the smooth normals using a low pass kernel.

To assist the user in defining the editing region, we use a segmentation method [35] which was extended [13] to handle RGBN images. The procedure is fast and is well suited to interactive applications. It also takes into account the normal and color channels, using all available information to improve results. It is also possible to segment objects based solely on geometry. To avoid over-segmenting, a bilateral filter should be used prior to segmentation to remove noise while preserving edges.

For replicating the RGBN sample on a base RGBN, we extended jump-map based

texture synthesis [27], more specifically jump maps on RGBN images [17]. This method is non-parametric and pixel-based, avoiding the need of a local parametrizations on normal maps. It does not produce the best quality results but it is an interactive technique, allowing for easy user experimentation. This strategy also handles base geometry distortions by varying image edge lengths during synthesis as a function of the normals.

During texture synthesis, colors are defined for each pixel. However the synthesized normals (high frequency) still have to be combined with the filtered input normals (low frequencies). We want to combine normals controlling which frequency bands we are replacing or editing. We follow the approach of [36], where two combination schemes were proposed: the linear model and the rotation model. The linear model combines normals by looking at them as gradients. This strategy is simpler and has the advantage that the resulting normals are guaranteed to correspond to a real surface, but the equivalent position displacements are restricted to the z-direction. On the other hand, the rotation model deforms the details to follow the base shape before combining. We use this method for all examples.

3.4 Jumpmaps on RGBNs

We begin this section by reviewing jump map-based texture synthesis on a regular image. We also describe a few principles that makes the method better suited to our application. We then discuss [17] the extension to color synthesis on RGBN images.

The simple example-based synthesis algorithms [28, 29] exhaustively search the input for a best match. This search is done for each pixel or patch being synthesized. In [27], Zelinka splits texture synthesis in two phases: analysis and synthesis. Note that in previous methods analysis was being done online 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 and store the similarities between all neighborhoods in the examplar. The Jump Map data structure holds for each

examplar pixel a list of *jumps* to similar pixels.

During synthesis, the output texture is traversed and pixels are sequentially copied from the example. Eventually the input texture border will be reached, that is when the jump map is most useful. 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 (NN) problem in a neighborhood space. Each pixel's square neighborhood is encoded in an M^2 feature vector, using the L_2 norm for comparisons. Notice that two parameters influence performance M and N (input sample size). Since analysis with small values of M will not represent the texture appropriately we are led 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 [37] that allows very fast queries. For further optimization, PCA is used for dimensionality reduction of the feature vector.

As Zelinka notices, jump map synthesis works best for stochastic textures and weak structured textures. While good results can be obtained with highly structured textures (Figure 3.11), these are exceptions. The reason can be seen in Figure 3.4. We see very similar neighborhoods, but if a jump is taken between them offsets will be introduced in the synthesized bricks. This problem happened because a small neighborhood was used for analysis. While increasing M improves results, it also increases analysis time.

For highly structured textures it is more important to match structure than to transfer details. We use multiresolution to ignore sample details during analysis. We notice that it is possible to use different values of N (different resolutions) for analysis and synthesis, virtually allowing larger masks. We use small samples for fast analysis, while we use detailed samples for quality synthesis (Figure 3.5). Zelinka uses multiresolution but only to improve PCA compression.

To extend this algorithms to RGBN images, two problems have to be solved: synthesizing normals and synthesizing **on** normal images. The next section presents our solution to the first problem. To handle the second one [17] adapted jump

Figure 3.3: Jump Map synthesis results.

Figure 3.4: The highlighted neighborhoods are similar, but a jump between them introduces offsets, breaking brick alignment. A bigger mask size (19×19) improves results.

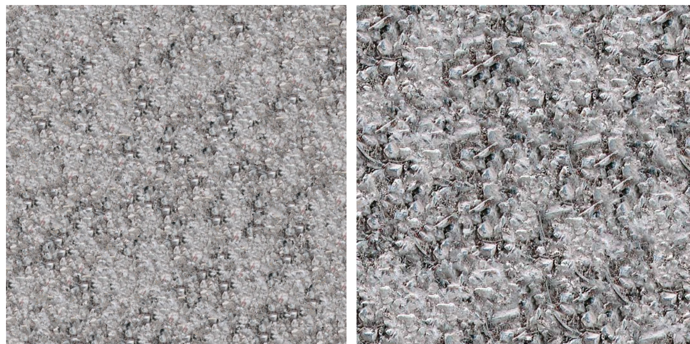
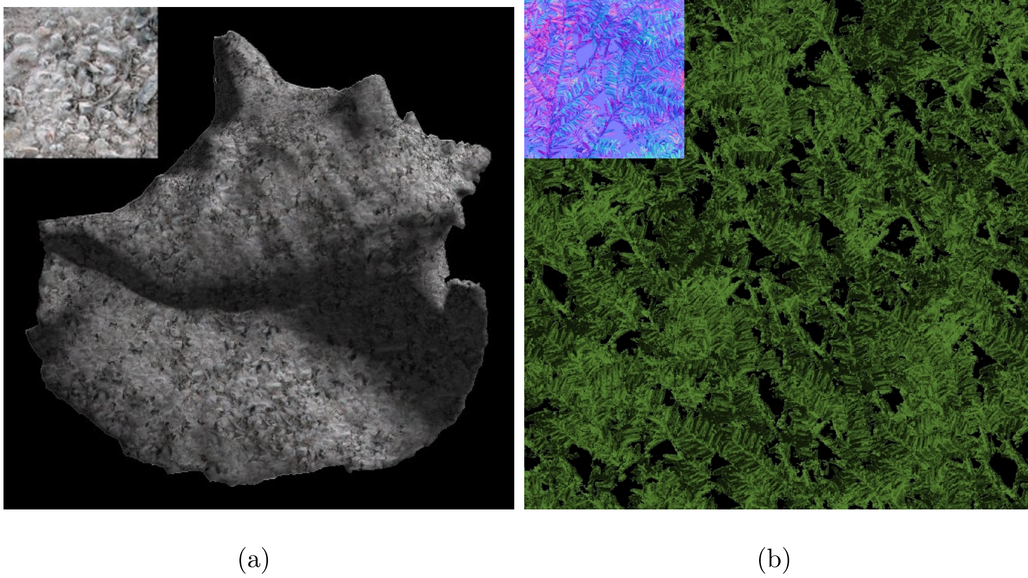


Figure 3.5: In both images, analysis was done with a 100×100 sample, but the second was synthesized with a 500×500 higher resolution sample, which would be hard to analyze.



(a) (b)

Figure 3.6: a) Due to foreshortening, texture scale varies spatially according to the normals. b) The leaves were replicated generating a normal map. We show here a shaded version of the synthesized map.

map-based color synthesis to normal images. On RGBN images, we need to vary the scale of the synthesis spatially due to foreshortening (Figure 3.6-a, Figure 3.7). A unit pixel offset in the output image induces a displacement in the represented surface, which in turns induces an offset in texture space. These parameters may be global or may vary smoothly over the surface [32]. 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 and of the projection p of d_i onto the surface $d_t = \frac{|d_i|}{|n_z|}\frac{p}{|p|}, p = d_i - < n, d_i > n$.

3.5 Normal Textures

We have extended the analysis and synthesis steps of the jump maps method to synthesize normals. In the analysis phase, we take a normal map sample representing our desired texture and we are asked to build a jump map for it. Since the basic primitive in jump map analysis is comparing neighborhoods, we must settle on

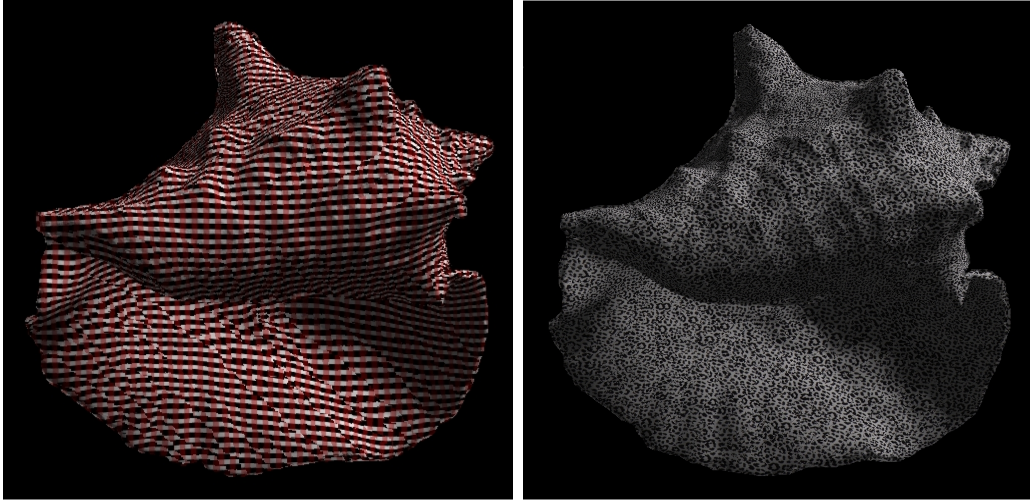


Figure 3.7: Different textures synthesized on a shell. Due to foreshortening, texture scale varies spatially according to the normals.

which metric to use. In the color setting, the sum of the euclidean metric for each pixel neighborhood was used. However normals are in the unit-sphere S^2 and as a consequence each neighborhood feature vector is in the cartesian product $S^2 \times S^2 \times \cdots \times S^2$. Therefore, the sum of the pixel normal metrics will induce a natural definition of a neighborhood metric, we only have to choose a normal metric. We analyze three different metrics: euclidean, geodesic and dot product based.

The geodesic distance obtained from the angle $\cos^{-1}(< n_1, n_2 >)$ is a natural choice (intrinsic distance) and it would be easy to adapt the $O(N^2)$ solution to the NN problem in the analysis phase. On the other hand, since cos^{-1} is a non-linear function, it is very hard to adapt either PCA or ANN, the fundamental algorithms that allow building jump maps in reasonable time. Linearity could be obtained by using a dot product based distance $1- < n_1, n_2 >$, but it falls far from the geodesic distance (Figure 3.8) specially for small values. In NN problems, these are the exactly the ones we are most interested in. This leads us to the last metric we consider: Euclidean. The Euclidean metric is a very good approximation for the geodesic metric for small distances since their derivatives at zero agree. Therefore, the Euclidean distance gives us a good compromise between simplicity and precision,

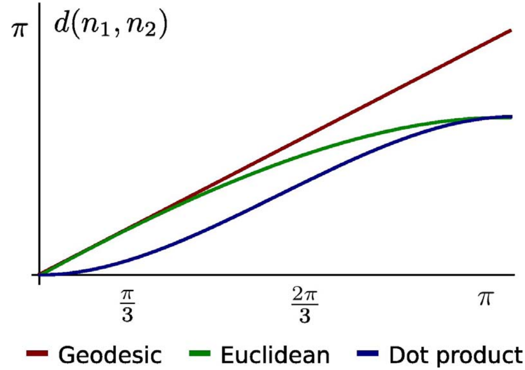


Figure 3.8: Different normal distances as function of the angle between the normals.

so this is our choice for texture analysis.

As for synthesis, the order in which pixels of the output image are synthesized is important to reduce directionality bias. This happens because each pixel is synthesized based on only one of its neighbors. Zelinka found that following Hilbert curves produces much better results than scan-line or serpentine traversals. However, we do not use Hilbert traversal. The reason is that we look for synthesis in subsets of RGBN images, which could be defined by either user or automated segmentation. Since these regions may have very complex shapes and topologies, it would be complicated to use a synthesis order based on Hilbert curves. Instead, we used a depth-first search (DFS) using a 4-connected pixel neighborhood. We recursively visit each pixels and its neighbors. The naive DFS algorithm produces unpleasant results with a directionality bias. This is a consequence of always visiting a fixed neighbor first, for example the north neighbor. A simple alternative is to use DFS, proceeding to each cardinal direction in random order. This breaks directionality and generates results almost as good as Hilbert traversal ones, as demonstrated by all results in this manuscript. The DFS traversal has two advantages. First, it only processes the pixels being synthesized. Second, it can respect discontinuities in the normals. For instance, in Figure 3.6-a we would not want synthesis to cross from the upper part of the shell to the lower part, we would like them to be in different connected components in the graph. This is easy to handle, if two neighboring pixels have different normals (defined by a threshold), they are simply not connected in

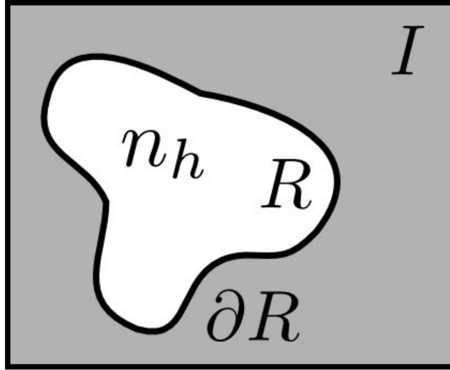


Figure 3.9: The detail normals n_h defined in R are combined with the image normals.

the graph.

3.6 Integrability Analysis

In this section, we discuss the conditions for the resulting normals to be integrable, that is, for it to correspond to a surface. For simplicity, we only analyze the linear combination method (Section 3.3) and disregard the foreshorten distortions introduced by scaling.

Given a base normal field n_b defined in the entire image I (Figure 3.9) and a synthesized detail normal field n_h defined in $R \subset I$, the problem of combining normals is generating a new normal field which agrees with n_b everywhere, but is influenced in R by n_h. We refer to their respective height functions as b and h.

We can look at a normal image n as a 2D vector field w (Section 3.3). If this field is the gradient of a height function, we say it is a conservative vector field. This means our normal image does correspond to a surface.

As seen in the previous chapter, conservative results can be guaranteed when combining normals by adding their vector fields counterparts w_h and w_b. Three conditions had to be satisfied. First of all, w_h and w_b had to be conservative. Second, since h is only defined in R, we must extend it as zero outside the edited region. This requires w_h to fall smoothly to zero close to ∂R and be equal to 0 outside. However, this is not enough. The added detail must not introduce level

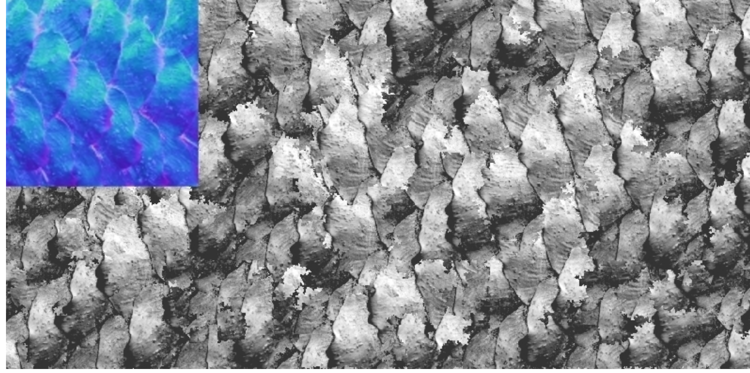


Figure 3.10: When the low frequencies of the exemplar are not removed, it is harder to infer pattern. This generates an uneven result.

changes outside R or it will create discontinuities in height.

What restrictions do we have to make to guarantee that normal synthesis satisfies the three requirements above? To begin with, w_h might not fall smoothly or not even fall to zero in ∂R, which would result in discontinuities in w. In many applications (Figure 3.11), w_b is already discontinuous in ∂R so that new discontinuities are not created. Second, synthesis will not introduce level changes if w_h only contains high frequency content which tends to oscillate and cancel itself. Hence, the restriction on w_h is enforced with the high-pass filter on the exemplars. In addition, texture synthesis methods in general, and jump maps in particular, do not introduce repetitions and so no low frequencies are created.

It seems very unlikely that non-parametric texture synthesis methods can guarantee the generation of conservative fields in arbitrary domains, since only local operations are performed. On the other hand, local operations seem enough to generate curl-free fields that guarantee that w is conservative under the additional hypothesis that R is simply connected (does not contain holes). Traditional techniques aim at generating texture such that each pixel's neighborhood closely resembles a neighborhood in the exemplar. We argue that the curl is also similar in this neighborhood. This means given curl-free exemplars, the synthesis will generate approximately curl-free textures. As Figure 3.11 shows, results of good quality can be obtained even when R is not simply connected.

3.7 Results

In this section, we will discuss some of the results obtained. Shaded images were produced with one light source. In some examples, uniform albedo is used to better highlight shape. In Figure 3.13-b, only the normals of the shell were changed, colors were unaffected. The base geometry was combined with high frequency normals extracted from rust. These new normals retain the original shape of the shell but give the appearance of a new material. It is true that, with some trial and error, rust could be generated with a procedural noise. On the other hand, structured synthesis from real objects on the shell (Figure 3.13-c,d) can only be accomplished with texture from example methods. These structured examples show how synthesized details follow the base geometry.

In Figure 3.6-b, we can see a normal map sampled from scanned leaves and synthesized normals. Uniform albedo was used for shading. There are some aliasing artifacts which could be handled by synthesizing in a higher resolution and down-sampling. This is possible, since synthesis time scales linearly.

In Figure 3.11-upper left, the original model contained a wooden board with real vegetables. Segmentation was able to separate the objects. We added the relief and color of the armor of a stone Chinese warrior to the board. The complex topology of this object was not a problem and DFS was able to guide synthesis around it. Rust was synthesized on the vegetables (normal and color). The fine scale normals on the normal map are hard to spot on the shaded version.

In Figure 3.10, a sample of the normals of a pine cone was used for synthesis. Low frequencies were not removed. This can be seen as the exemplar varies smoothly from light (top) to dark blue (bottom). It has two consequences, first it is harder for the analysis phase to infer pattern. Second the final result is uneven. The shaded image shows some darker regions which are not facing the light.

As in Textureshop [12], shape from shading can be used to recover normals from photographs. We have also used it to obtain the texture exemplar (Figure 3.12). A swatch was extracted from the lizard's arm (b) and combined with smooth hand

Figure 3.11: A shaded image of edited vegetables. Automatic segmentation followed by texture synthesis is a powerful tool for material replacement. The vegetables were turned into rust metal.

normals (d). The final image (e) was shaded under diffuse lighting, ignoring the lizard's skin reflectance properties. For more realistic results, a synthesis method that can handle morphing between textures is required. Notice how the texture of the fingers, the hand and the arm of the lizard are different. Very high frequency detail like human hand lines could be recombined in the final result.

3.8 Conclusion

The presented method has its limitations. First, requiring the deformations to extend to 0 outside R is one of them, blending techniques should be investigated in the future to enforce this property on any synthesized deformation. Additionally, a limitation of jump map-based synthesis is that being pixel-based, it has problems with very structured textures. On the other hand, it is harder to adapt less local methods to RGBN images, since the metric of bigger neighborhoods is non-trivial. In future work, we would like to use normal synthesis in the context of inpainting normals to fill missing regions.

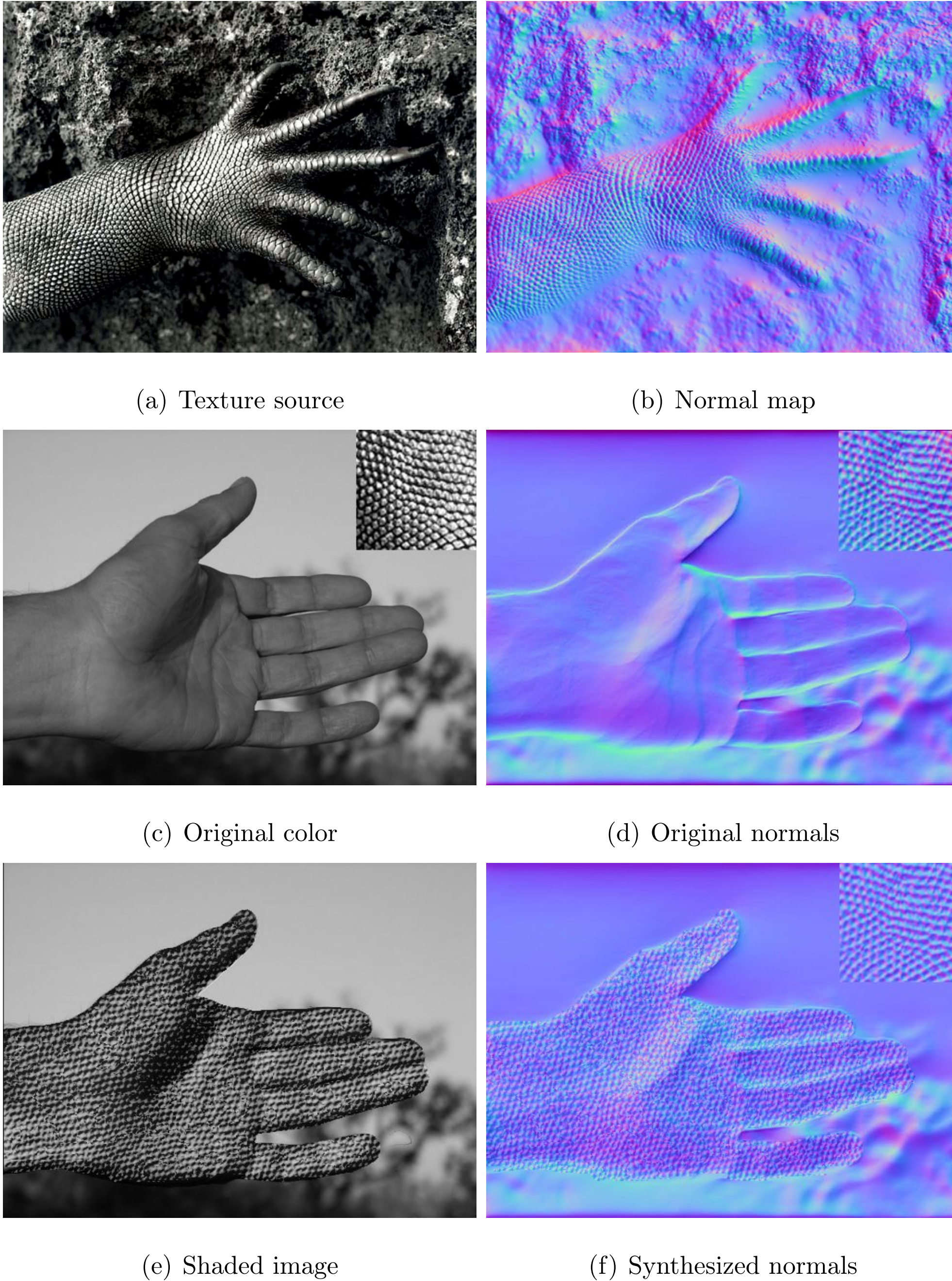
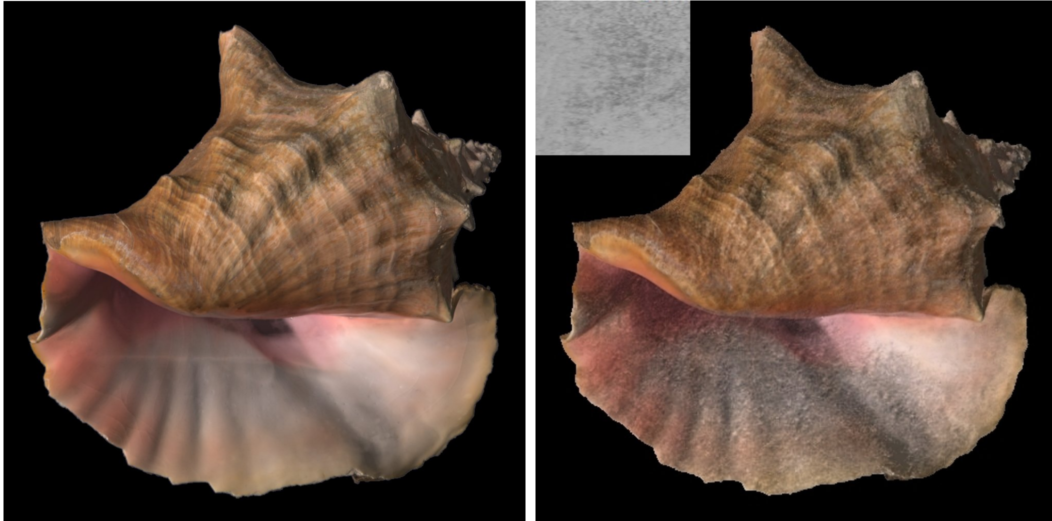
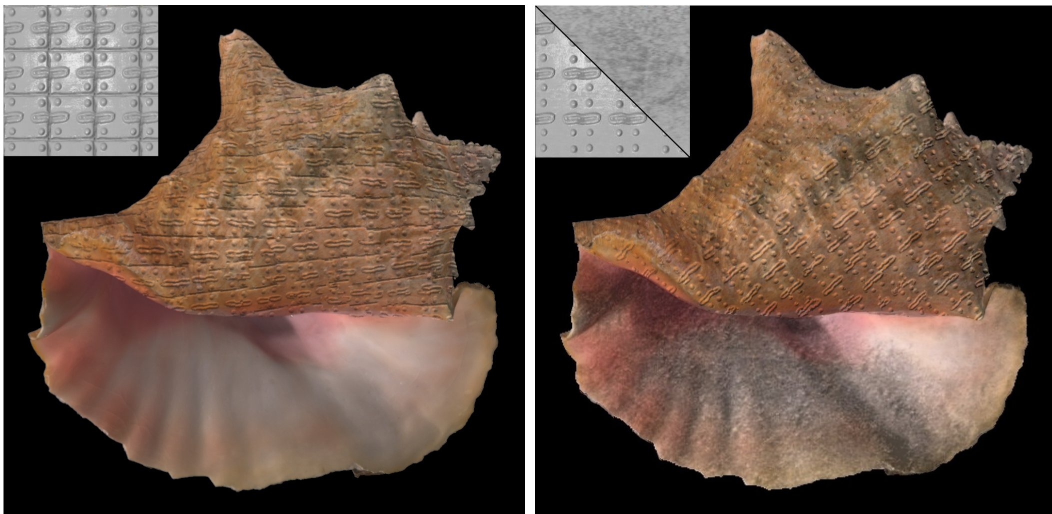


Figure 3.12: Shape from shading was used to estimate normals (b,d). A small sample of the lizard's arm was filtered and synthesized over the hand (e). Lighting (e) was set similar to the original. Only the normals were used to produce the final result.



(a) Original Shell (b) Rust Normals



(c) Armor Normals (d) Armor Normals

Figure 3.13: The small images show the exemplars used under a direct light source. Images are embedded in high resolution. The last examples show structured texture synthesis.

In this chapter, we have developed a method to synthesize normal textures on RGBNs. Our pipeline includes normal filtering to control which frequency bands we edit or replace. The next chapter presents in an abstract level our surface normal editing process and how the RGBN operations are integrated using a projective atlas [38].

Chapter 4

Chart Editing

In the last chapters, we presented many operations with normal vectors using the RGBN image representation. The main limitation of RGBN image is that they are limited to a single view point. In this chapter, we present a solution for mesostructure editing represented by surface normal vectors. First, in Section 4.2 we detail the separation of geometry into base surface and details. This separation is fundamental for our surface representation described in Section 4.5. While the base surface is stored explicitly, the details are only represented using normal vectors in a texture atlas. Second, we discuss the limitations of the texture atlas for editing and present our editing framework. We use additional temporary charts for editing. In fact, we use ortographic projective temporary charts which have advantages regarding normal processing.

4.1 Related Work

There are multiple representations and algorithms for modeling geometry interactively. There are methods that represent geometry directly with a triangle mesh. Multiple methods have been developed that map free-form user edit specifications to mesh deformations. Some methods pose the specifications as boundary conditions and use some form of Laplacian preservation to propagate changes [39, 40, 41]. Other methods build on this framework but handle details better, either encoding

details in local frames [42] or searching for locally rigid transformations [43]. These approaches have limitations regarding the resolution of the models they operate on. The reason is they represent both macro and mesostructure in a triangle mesh, which is a non-hierarchical model. Therefore they cannot accommodate multiscale edits and must solve for a deformation across all scales simultaneously, usually leading to large linear systems or optimization problems. But the main problem with meshes is their lack of regularity.

In contrast to irregular meshes, subdivision surfaces [44] have an irregular control mesh but are regular after a few subdivision steps. These surfaces easily represent smooth models with few control points and details with a large number of control points. Similar to meshes, they do not include a multiresolution editing semantics. There is no way of changing the mesh broadly while maintaining details.

These limitations led Zorin et al. [1, 45] to propose the multiresolution surface, i.e, a multiresolution representation based on subdivision. It is composed of a base surface and details stored in local coordinate systems at different levels of resolution allowing multiresolution editing. Similar multiresolution surfaces have been used to produce highly detailed surfaces in commercial packages such as ZBrush [16] and Mudbox [46]. Instead, we do not employ a full multiresolution decomposition, we only decompose the surface in two scales: macro and mesostructure.

Some works on subdivision surfaces have used parametrizations for operations. Ying et al. [47] propose a local walk around a vertex to build a local parametrization. This neighbourhood is then used for texture synthesis. Biermann et al. [48] present a solution for transferring details between meshes. During an editing session, they parameterize both source and target surface subsets in a common planar domain.

There is one problem with subdivision and multiresolution surfaces. This representations are not invariant by fractional translations. As a consequence if we want to move a given detail on the surface, we will have to change all the representation coefficients across many scales. For example, placing a specified feature on any position on the surface may be non-trivial. [49].

A solution to this problem is maintaining an irregular mesh as a base mesh, but

parametrizing it to obtain a regular representation for details. Many solutions have been introduced to parametrize a mesh [50]. In particular, some methods involve first partitioning the mesh and parametrizing it into multiple charts to reduce distortions. Some works use the parametrization to synthesize attributes in texture space [47, 32]. While most approaches work with both a mesh and a parametrization, it is possible to throw the mesh away completely. In geometry images [3], the authors build a parametrization and store position in the planar domain. They can render the mesh by decomposing this domain in microtriangles and rendering each one separately. Their approach has been extended to multiple charts [51] to reduce distortions.

We follow the approaches based in a mesh together with a texture atlas. While any texture atlas can be used in our method, we have chosen to use a projective atlas [38] since it was designed to store captured attributes. Unlike unparametrized meshes and subdivision surfaces, a model together with a texture atlas let us use different but appropriate representations for macro and mesostructure. Most works that use parametrizations do not take advantage of it for operating and storing geometry. The reason is as geometry changes, so should the atlas' parametrizations. The key ingredient of our method is its separation of scales. We assume the parametrizations are defined by the macrostructure which remains fixed, while we only change its mesostructure.

In our work, we decompose the model into a base surface and details, or macro and mesostructure. Multiple decompositions of this kind have been done before. These can be classified according to the detail representation into scalar and vector-based approaches. Blinn [4] first proposed to think of the surface as a smooth base enhanced with a scalar displacement in the base normal direction when he presented bump-mapping. Both the surface and the displacements were stored, but he only used displacements for rendering. On the modeling side, scalar displacements have been used in the work of Lee et al. [52], which formally defines their model as a smooth subdivision surface together with a scalar displacement. Besides rendering, they present methods for compression, editing and animation in this hybrid representation. Zatzarinni et al. [53] present additional applications of the hybrid

representation, such as, relief segmentation, detail exaggeration and dampening and detail transfer. As opposed to Blinn, these two works propose how to actually decompose a surface.

In addition to scalar methods, there are vector-based methods. Instead of storing a displacement in the base normal direction, these techniques store an arbitrary 3D vector [48]. Krishnamurthy [54] proposed to fit a dense polygon mesh with smooth B-splines and store vector displacements to recover the true surface.

In our work, we assume the surface can be described by a scalar displacement in the normal direction. However, instead of working with the scalar function itself, we only store the normals of the displaced surface, similar to the approach of Cohen et al. [7]. They simplify a mesh and store a normal map for faster rendering while preserving appearance. While this normal mapping approach and the original bump mapping approach [4] employ this separation with a rendering point of view, we propose methods to **process** this representation by formally defining normal operations. In this way, we bring this hybrid representation to modeling applications. This decomposition choice helps with maintaining a uniform compact representation over many modeling related stages including capture and rendering. During capture, it is common to acquire photometric normals in addition to smooth geometry. In interactive rendering, normals are used to enhance smooth models with details with little computational overhead. When normals are not enough for accurate rendering, the normals output by our system could be integrated to calculate a displacement map, but we leave this as future work. Even in this context, the different semantics of normal operations could be useful for modeling. Normal operations have been proposed before in Normalpaint [19]. However, besides being limited to extrusion of simple shapes, their method cannot process existing normals only create normals from scratch.

Having chosen textures to represent our details, we should go into the details of texture painting/processing methods, but we defer an in depth discussion to the next chapter. Most previous methods have been restricted to working with color attributes or at most scalar displacements treated like colors [55, 56, 57].

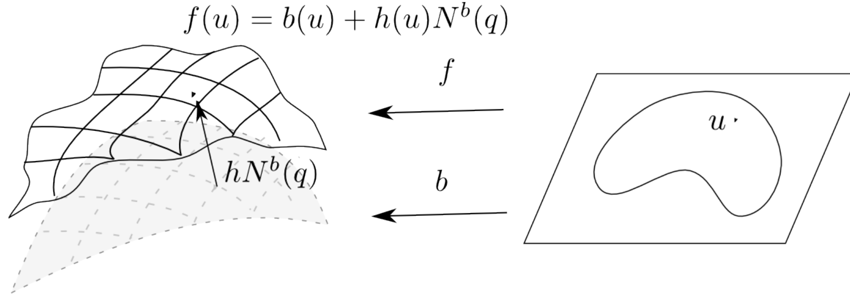


Figure 4.1: The modeled surface consists in a smooth base surface b displaced by a scalar field h in its normal direction N^b.

4.2 Scale Decomposition

The separation of macro and mesostructure is a key point in this work. Since we propose a solution for editing the mesostructure or details, we assume the macrostructure is fixed. In the next sections, we will see how this assumption can be used to design more efficient editing methods. In this section, we formalize our hybrid surface representation, i.e our separation of base surface and details. In addition, we define operations on this representation. For simplicity, we discuss the case of a single parametrization defined in a planar domain $M \subset R^2$.

There are multiple ways of defining this separation. We follow the original definition by Blinn [4] (Figure 4.1). In his work, the final surface $f : M \to R^3$ consists in details added in the base surface's $b : M \to R^3$ unit normal direction $N^b : M \to S^2$ by a scalar height function $h : M \to R$:

$$f(x) = b(x) + h(x)N^b(x)$$

However, unlike previous methods, we do not store the scalar displacement h, we only store the base surface b and the unit normal N^f of the original surface f (Figure 4.2). Notice that this information is not enough to easily obtain f. But we know that for each known normal field N^f there is an unknown scalar displacement h. As we have discussed before, working with normals brings additional advantages. Overall, our representation consists of the base surface b and the detailed normals N^f.

One important question at this point is how general this representation is. The

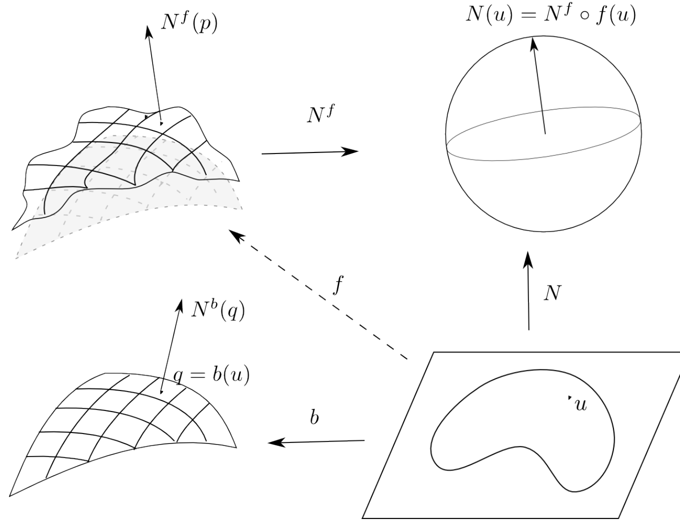


Figure 4.2: In addition to the base surface positions b, we store the original surface normals N^f. We do not have the surface f explicitly, it is implicitly defined by its normals.

base surface cannot be arbitrary. Previous works [52, 53] have shown how to find it in a way that the details can be described by a displacement. Even if we start with a proper surface represented, we must still guarantee that changing the details does not result in improper behavior like self-intersections. This control can be accomplished by limiting the possible values of h. The range of possible values of h define what is known in differential geometry as a tubular neighborhood [25]. In fact, given a C^2 base surface, we can guarantee that f is a surface for small values of h. More specifically, h can be as large as b's local feature size, i.e the distance from the surface to its medial axis. These results reinforce our thesis that when working with a macro/meso separation, it is best to work with smooth base surfaces, allowing for larger possible representable details.

We must make some remarks regarding notation. In this work, we will frequently overload notation when referring to functions defined on surfaces or to their expressions in the parametric domain. In addition, we will no longer refer to unit normals, we will simply write normal. In all of this work, normals are assumed to be unit vectors. Finally, we will often use $N = N^f$ to simplify notation.

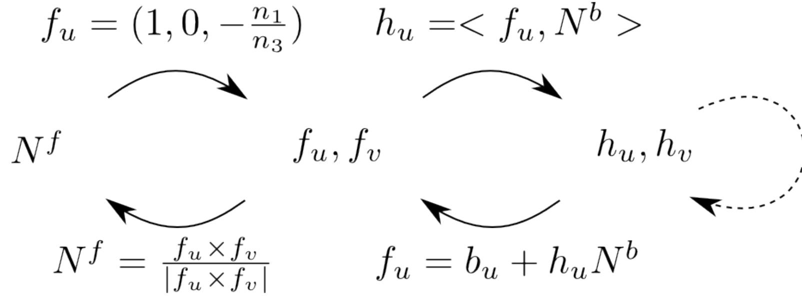


Figure 4.3: Normals are converted to scalar displacement gradients. Operations are performed in the gradient domain denoted by the dashed line.

4.3 Detail Processing

We store the normals $N = N^f$ because they are a common representation in both acquisition and rendering methods. However, just like in the RGBN image case, we can profit from converting normals to a gradient domain for modeling operations. In this section, we present the conversions between surface normals and scalar detail gradients. In addition, we show how to integrate them with the operations proposed in previous chapters. We will assume we are working in an orthogonal chart such that z is the up direction and the parametric coordinates u, v coincide with the x, y directions. Any other orthogonal chart can be transformed to this one by a change of basis. In fact, this change of basis must be performed when implementing the method.

We start by presenting the conversion between the normal N and the detail gradient h_u, h_v (Figure 4.3). First, we convert from $N = (n_1, n_2, n_3)$ to the parametrization tangent vectors f_u, f_v. Under this parametrization, the surface f can be written as $f(u, v) = (u, v, z(u, v))$, whose tangent vectors are $f_u = (1, 0, z_u)$ and $f_v = (0, 1, z_v)$. We can calculate these vectors using the identities $z_u = -n_1/n_3$ and $z_v = -n_2/n_3$, as we have seen in Chapter 2. In addition, we can obtain the base surface's tangent vectors b_u, b_v from N^b in the same way.

Next we convert to h_u, h_v. Recall that $f = b + hN^b$, as such $f_u = b_u + h_u N^b + h N^b_u$, analogously for v. We can extract h_u by taking dot products on both sides with

the base unit normal N^b and using the orthogonality between normals and tangent vectors:

$$\langle f_u, N^b \rangle = \langle b_u + h_u N^b + h N_u^b, N^b \rangle$$

$$\langle f_u, N^b \rangle = \underbrace{\langle b_u, N^b \rangle}_{0} + h_u \langle N^b, N^b \rangle + h \underbrace{\langle N_u^b, N^b \rangle}_{= \frac{1}{2}(\langle N^b, N^b \rangle)_u = 0}$$

$$\langle f_u, N^b \rangle = h_u$$

All RGBN image operations previously presented can be used to manipulate the detail gradients h_u, h_v including combination, filtering and warping. This is important because we can change only the details h keeping the base surface b untouched. During a gradient operation, it is important to bound operations, as discussed before the representation is not well defined for values of h larger than the local feature size.

After an editing operation is complete, we must update our representation. Therefore, we must convert from gradients to normals. First, we convert to tangent vectors using the formula $f_u = b_u + h_u N^b + h N_u^b$. However, we do not have the scalar displacement h, only its edited gradient h_u, h_v. As an approximation, we assume it null and discard the height dependent term using:

$$f_u = b_u + h_u N^b$$

Analogously for v. After obtaining the new tangent vectors, we return to $N_f = \frac{f_u \times f_v}{|f_u \times f_v|}$. This completes an editing cycle. At this point, we must justify the validity of the last approximation. Blinn [4] argued that for small values of h, such as those used in bump mapping, this approximation is valid. However the neglected terms hN_u^b, hN_v^b depend both on the scalar displacement h and on curvature related terms N_u^b, N_v^b. Therefore, keeping the error under control involves limiting both of these quantities. This is not a problem in our representation since we do assume small displacements and a smooth base surface. In summary, small h and smooth b helps us solve not only collisions with the medial axis but also let us work with normals only.

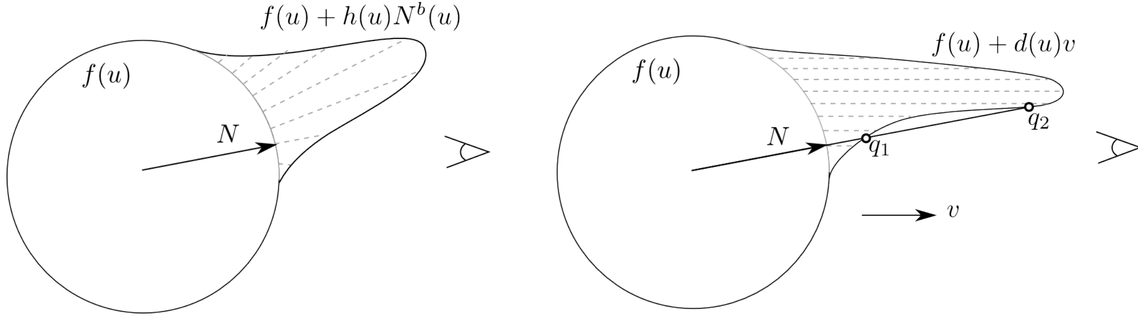


Figure 4.4: View-dependent editing consists in pulling the surface in the camera direction. However, large edits may result in a surface f which cannot be represented as a displacement of the base surface b. As shown, points q_1 and q_2 would me mapped to the same base point.

In Chapter 3, we presented a method to extract details from RGBN exemplars. We used linear filtering to extract the low frequencies and preserve the details. The problem with that approach was that the details were regarded as displacements in the z direction. The conversion from normals to displacement gradients in this section provides a more powerful alternative. We can filter the original RGBN image to define the base normals and use the above equations to extract the gradient of a displacement detail. In future work, we would like to use parametrization methods to extend this detail extraction method even further. While the above method compensates for the base tangent plane orientation, parametrizations can be used to compensate for the base surface metric distortions. After building a parametrization, the RGBN image warping methods of Chapter 2, could be used to transfer the detail gradients h_u, h_v to the new domain.

The method presented in this section falls back to the RGBN image processing method of Chapter 2 simply by considering the image plane as base surface $b(u, v) = (u, v, 0)$, thus $N^b = (0, 0, 1)$. In this setting $(h_u, h_v) = (z_u, z_v)$ as previously used. With RGBN images, since N^b is constant, we are in fact deforming the surface in a fixed direction. In the next section, we discuss fixed direction manipulation in the context of any base surface.

4.4 View-Dependent Processing

Displacements in the normal direction are well behaved (Figure 4.4). However, they impose restrictions on the possible operations. One useful operation is deforming the surface in a camera's direction w instead of the base normal direction. This is currently available in mesostructure commercial packages. The above method can be easily adapted to handle these deformations. Instead of working with scalar displacement gradients h_u, h_v, we simply work with height gradients z_u, z_v. Operations proceed similarly. In fact, this view-dependent processing is just like the RGBN image operations previously proposed. Given a parametrization $f(u, v)$ and a displacement $d(u, v)$, we can write the edited surface as:

$$g(u, v) = f(u, v) + d(u, v)w$$

While for normal displacements, the tubular neighborhood theorem guaranteed we had a surface and gave us sufficient conditions to control our operations, the situation is more complex for view-dependent editing. For instance, we can prove that the result is a surface. However, the resulting surface may not be representable as a scalar displacement from our base surface (Figure 4.4). In other words, we may be required to change the base surface, which we leave as future work. In practice we have no problems for small edits.

To prove that we are working with a surface it must be shown that $g(u, v)$ is differentiable, an homeomorphism and has a well defined tangent plane. Differentiability of g follows from the differentiability of d. It can be shown that it is an homeomorphism assuming d does not cause self-intersections and using the fact that all displacements are parallel in the w direction. Now, we show that it has a tangent plane by showing that we can define a normal field $g_u \times g_v$ that does not vanish. We claim that for a given d this holds if and only if the camera direction w is not in the span of f_u, f_v for any edited point. Expanding $g_u \times g_v$, we get:

$$g_u \times g_v = f_u \times f_v + d_v f_u \times w + d_u w \times f_v$$

In other words, we have to prove that there exist d_u, d_v such that $f_u \times f_v + d_v f_u \times w + d_u w \times f_v = 0$ (1) if and only if $w = c_1 f_u + c_2 f_v$ (2). By assuming (2) and

substituting in (1), we get:

$$f_u \times f_v + d_v f_u \times (c_1 f_u + c_2 f_v) + d_u (c_1 f_u + c_2 f_v) \times f_v = (1 + c_1 d_u + c_2 d_v) f_u \times f_v$$

We can choose d_u, d_v such that the linear coefficient is zero and thus $g_u \times g_v = 0$, proving that (2) implies (1).

On the other hand, assuming (1) holds and rewriting we get:

$$f_u \times f_v = -d_v f_u \times w - d_u w \times f_v = (-d_v f_u + d_u f_v) \times w$$

But cross products have the property that $c = a \times b \rightarrow c \perp b$. Therefore w is orthogonal to $f_u \times f_v$, as such, it must lie the tangent plane spanned by f_u, f_v. This proves that (1) implies (2), completing the demonstration that we have a surface.

All the preceding discussion, considered w to be the camera's direction. However, this is not necessary, the above demonstration would work for any arbitrary fixed direction q. While editing in the direction w is defined for all points in the orthogonal chart, editing in direction q will only be defined in a subset of the visible pixels. The reason is the impossibility of aligning q with the base surface's tangent plane. While editing in the camera direction is a natural extension of RGBN image editing, we leave editing in an arbitrary direction as future work.

4.5 Atlas Representation

The previous section proposed a mathematical representation of surfaces and operations on these objects. In this section, we present the discrete models and algorithms. It is only in the next chapter that we present implementation details.

There are many possible representations for surfaces in modeling, for example, polygon soup, parametric surfaces, subdivision surfaces. Ideally, we would like a manifold representation [58, 59]. It would let us work in a differentiable chart for any point on the surface. However, building such a representation is a hard problem. One of the main problems is that while in differential geometry the manifold representation defines a fixed surface, in modeling, surfaces are changing all the time.

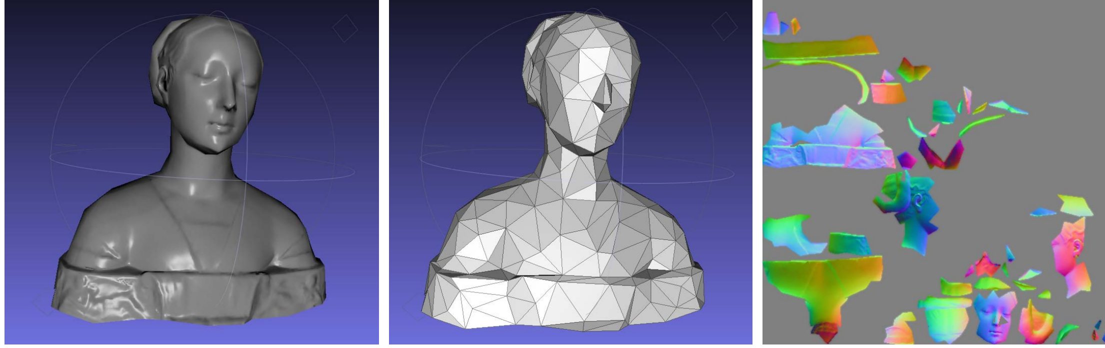


Figure 4.5: We use a texture atlas with two attribute functions defined: fine scale normals (right) for details and coarse scale positions (center) for the base surface.

The way we approach the problem of changing surfaces in this work is through a separation of the surface in macro and mesostructure. As our problem is editing mesostructure, we can assume the base surface is fixed. In other words, we can edit geometry while keeping it fixed.

Most 3D surfaces are not globally homeomorphic to open sets in the plane. For this reason, we cannot have a single mapping from the surface to texture space. Inspired by differential geometry, the graphics community has approached the texture problem as an atlas construction problem [60]. We chose this solution. For it to work, we additionally assume the topology of the surface is the same of the base surface. The fixed topology of the base surface lets us represent our surface using a texture atlas (Figure 4.5). This way we can work with objects of complex topology by splitting it in pieces that can each be mapped to planar regions. Mathematically, geometry will be represented by functions on the charts. We require two attribute functions defined on this atlas: fine scale normals for the details and coarse scale positions for the base surface. In addition, the texture atlas representation is appropriate for storing other attributes like diffuse albedo and specular albedo.

The main difference between the manifold representation above and the texture atlas has to do with chart overlaps. Since some of the surface attributes are by definition its values on the charts, it is very important that we keep the charts disjoint and avoid conflicts in the definition of attributes. In other words, two

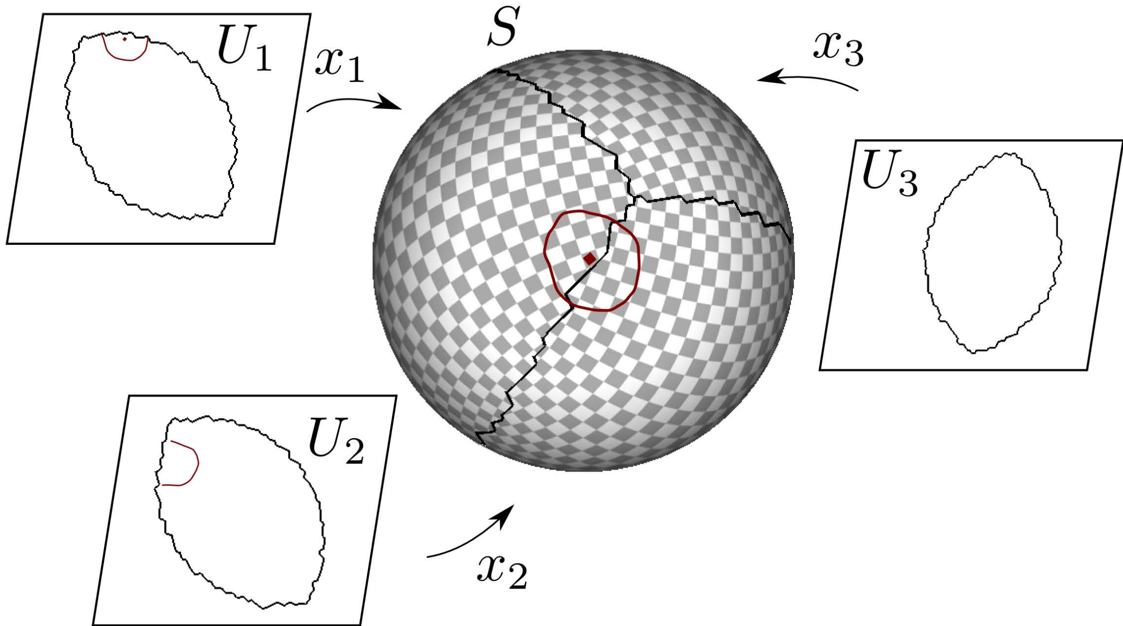


Figure 4.6: Surfaces are decomposed into an atlas.

different charts containing the same surface point could store different attributes for this point. From now on, we shall refer to this texture atlas as a storage atlas (Figure 4.6). Next we formalize these concepts.

Definition 1. *A **storage atlas** of a surface $S \subset R^3$ is a set of pairs $\{x_i, U_i\}$, called **charts**, such that $x_i : U_i \to S$ are C^2 difeomorphisms between the sets $U_i \subset R^2$ and their image on the surface with the following properties:*

- $\bigcup_i x_i(U_i) = S$
- $x_i(U_i) \bigcap x_j(U_j)$ *is either a set of curves and points or the empty set*

This definition means that S is the disjoint union of the chart images except for edges or isolated points. We can thus refer to each point on the surface as being contained in a single chart image $x_i(U_i)$. We can think of a bijection between S and the set $\bigcup_i U_i$, except for chart borders. Any function defined in the disjoint sets U_i will induce a function defined on the surface. In other words, in our discrete setting, a **surface function** f on a surface S is defined by its expression in the disjoint

charts $\{U_i\}$. We will also use the names surface **attribute** or surface **signal** to refer to a surface function.

The storage atlas described above is very common in computer graphics representations. There are some desirable attributes of an atlas, most of them are related to how quantities are measured in the planar domain versus on the original surface. For example, we would like parametrizations to preserve angles, areas or lengths [50]. The graphics community has proposed many algorithms for splitting a given surface and constructing the planar mapping of each region. Any storage atlas could be used in the editing process proposed in this work.

4.6 Editing Chart

Recall that the storage atlas is a disjoint set of charts so that attributes are defined with no redundancy. On the other hand, it is inconvenient for editing attributes. If we were to filter a color signal on the surface for example, we would evaluate an integral over each point's neighborhood. The problem with disjoint partitions is that a neighborhood of a point may not be contained in a single chart (Figure 4.6). One trivial but incorrect solution is to use only the fraction of a point's neighborhood contained in its chart. However discontinuities are usually created when editing in this way. In Figure 4.7, we see the consequences of this discontinuities, also called seams, for both color textures and displacement maps.

A trivial example in which the surface is planar is shown in Figure 4.8. A Gaussian blur filter was applied to the color attribute for noise removal. Compare the result of filtering with a single chart encompassing the image versus filtering with two charts without overlaps. A discontinuity is created when we use two disjoint charts of the plane, one for each half of the image. This approach introduces a discontinuity on the middle of the image.

In [61], the authors place seams in low visibility regions. Consequently, any discontinuities in texture will be seldom noticed. Piponi [62] overcomes the discontinuities by using localized overlapping charts along the seams. Actually, he only

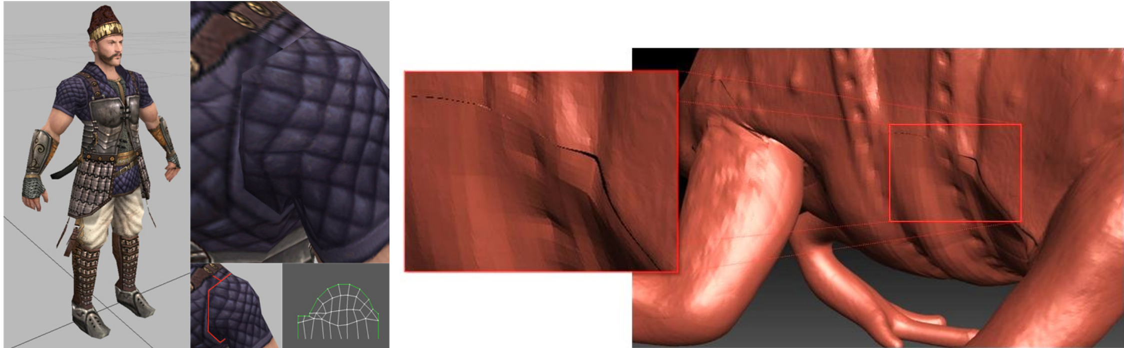


Figure 4.7: Editing with disjoint charts introduces discontinuities, also called seams. Examples shown for both color textures and displacement maps.

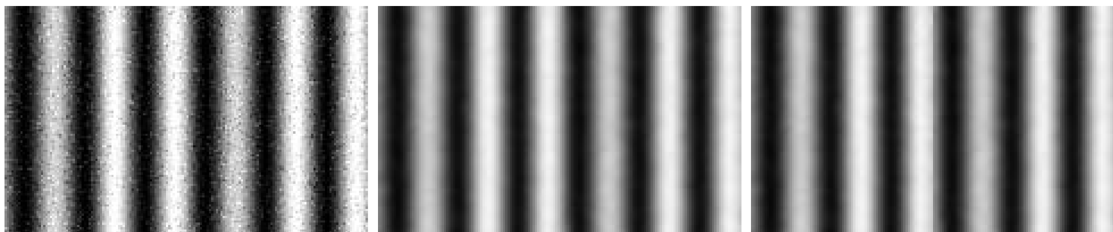


Figure 4.8: A Gaussian blur filter was applied for noise removal. Compare the result of filtering with a single chart encompassing the image versus filtering with two charts without overlaps. We used a chart for each half of the image. This approach introduces a discontinuity on the border.

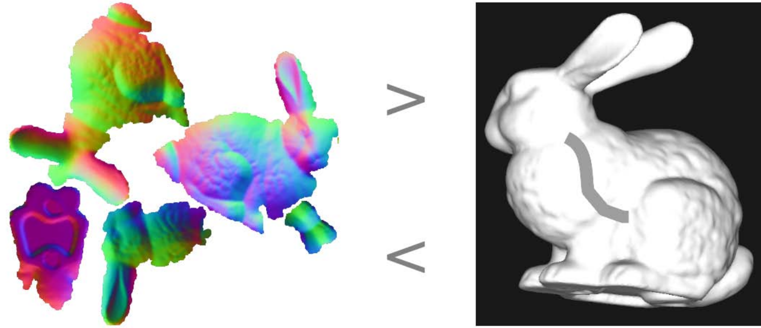


Figure 4.9: We project attributes, creating the temporary chart. All operations are performed in this chart. Finally, all changes are mapped back to the original atlas.

uses two charts. One chart parameterizes the entire surface, except for a connected set of seams. The second chart is defined in a small neighborhood around the seams. However, they are not disjoint, the two charts overlap. The author uses a partition of unity to blend textures defined in different overlapping charts. His technique could be directly extended to multiple charts. One problem with his approach is that artists have to paint the multiple charts separately. He leaves as future work, the development of a painting interface that updates multiple charts simultaneously.

We also use an overlapping chart. In addition to the disjoint set of storage charts $\{x_j, U_j\}$ defined above, we introduce a temporary editing chart $\{x, U\}$. Editing with the temporary charts consists of building the editing chart, processing the attributes $\{f(U_j)\}$ and finally updating storage charts (Algorithm 1). A localized editing operation (Figure 4.9) starts by building an editing chart around the region of interest, such that the region is completely contained in the editing chart. Since the storage charts cover the whole surface, the editing chart will intersect some of them. Attributes are transfered from the intersecting charts to the editing chart (Algorithm 2) using transition functions $\{x_j^{-1} \circ x\}$. This results in an attribute function g defined in U. At this point, we can perform the operation itself, e.g. painting or filtering. After the operation is finished, the attributes are mapped back to the storage charts (Algorithm 3) using $\{x^{-1} \circ x_j\}$. This process is detailed in the algorithm below:

Input: Editing chart $\{x, U\}$, storage atlas $\{x_j, U_j\}$, surface signal $\{f(U_j)\}$

Output: Processed signal

Build($g(U)$ from $\{f(U_j)\}$);

Process(g in U);

Update($g(U)$ to $\{\bar{f}(U_j)\}$);

Copy($\bar{f}$ to f);

Algorithm 1: Temporary chart processing algorithm

Input: Editing chart (x, U), storage atlas $\{x_j, U_j\}$, surface signal $\{f(U_j)\}$

Output: Signal loaded in chart (x, U)

```
foreach Storage chart U_j do
    if x(U) ⋂ x_j(U_j) = W ≠ ∅ then
        foreach p ∈ x^{-1}(W) do
            g(p) := f(x_j^{-1} ∘ x(p));
        end
    end
end
```

Algorithm 2: Building editing chart algorithm

Input: Editing chart (x, U), storage atlas $\{x_j, U_j\}$, surface signal $\{f(U_j)\}$

Output: Storage atlas $\{x_j, U_j\}$ updated

```
foreach Storage chart U_j do
    if x(U) ⋂ x_j(U_j) = W ≠ ∅ then
        foreach p ∈ x^{-1}(W) do
            f(x_j^{-1} ∘ x(p)) := g(p);
        end
    end
end
```

Algorithm 3: Update storage charts algorithm

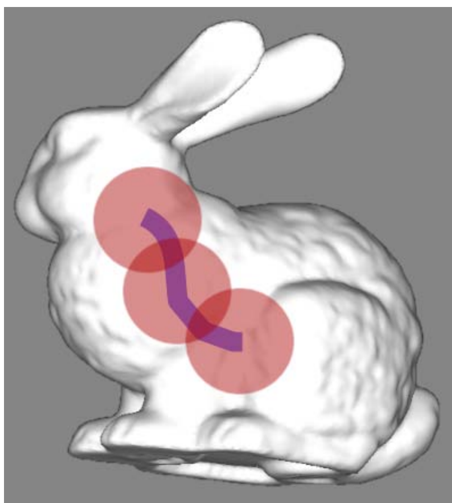


Figure 4.10: The image itself is used as an editing chart. The editing region shown in blue is fully contained in the editing chart. In fact, large neighborhoods of the blue points are still contained in the chart. These neighborhoods are shown in red.

The main advantage of the editing chart is that not only the region of interest will be contained in it, but also any point in the edited region will be sufficiently far from the editing chart's border. As a consequence, a sufficiently big neighborhood of each editing point will be fully contained in the chart. For example, in Figure 4.10 the image itself is used as an editing chart. The editing region shown in blue is fully contained in the editing chart. In fact, large neighborhoods of the blue points are still contained in the chart. These neighborhoods are shown in red. This characteristic is very important for filtering. For example, in Figure 4.11, we used Gaussian filtering to remove lighting differences between charts. The middle image presents color filtered independently in each storage chart, which preserves seams. The final image presents the result of filtering color using editing charts. By respecting the object's topology, color leaks between the charts and leads to a seamless result.

Another big advantage the editing chart shares with any other texture based technique is its simple topology. Almost all pixels are arranged in a grid and have easily defined 4 or 8 connected neighborhoods or even larger neighborhoods. For instance, this is very important for filtering, in which a convolution will be performed.

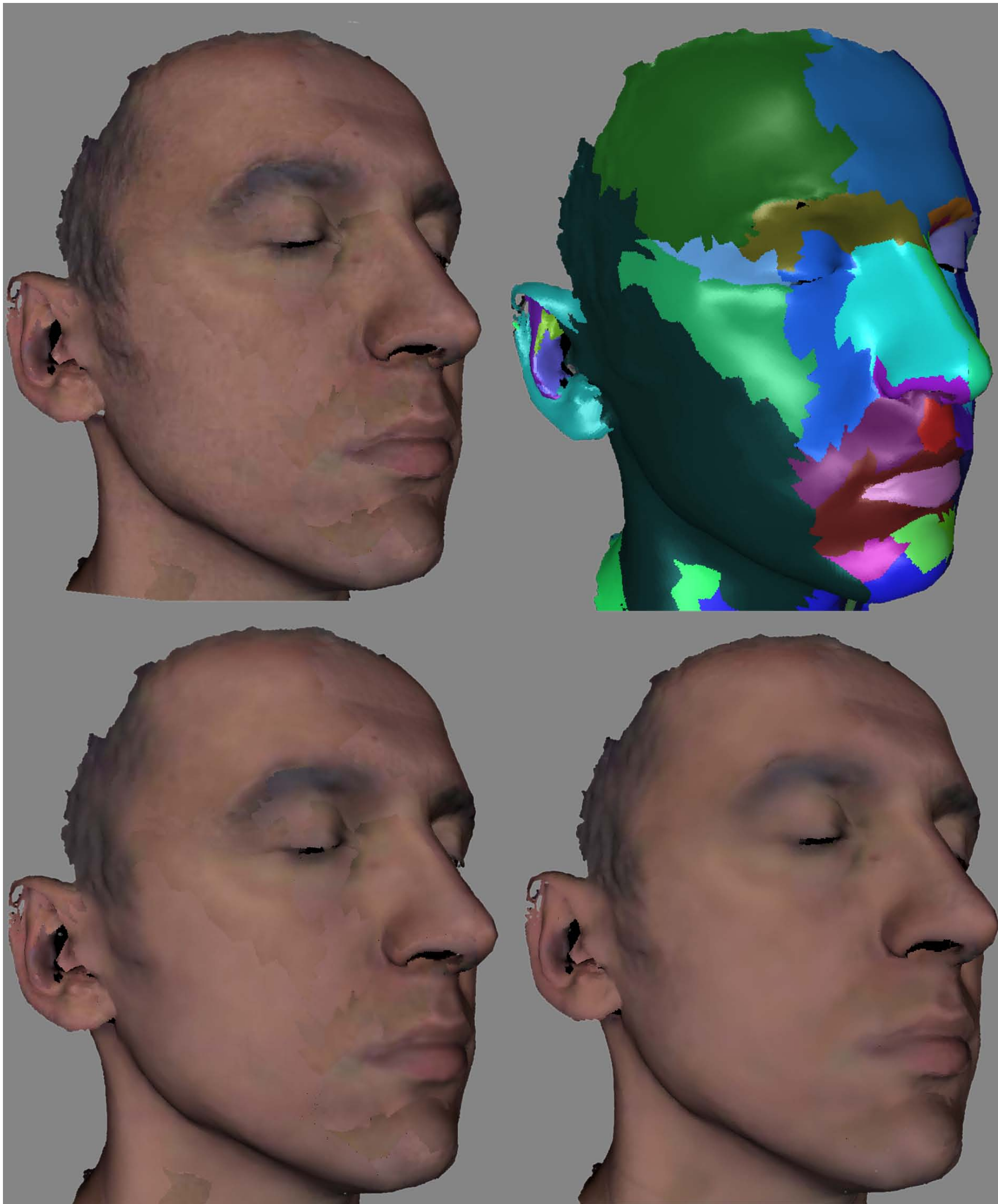


Figure 4.11: In this example, we used Gaussian filtering to remove lighting differences between charts. The first image shows the chart structure followed by the photographed colors. In the bottom left, we filtered the signal independently in each storage chart. The final image presents the result of filtering color using editing charts. By respecting the object's topology, color leaks between the charts and leads to a seamless result.

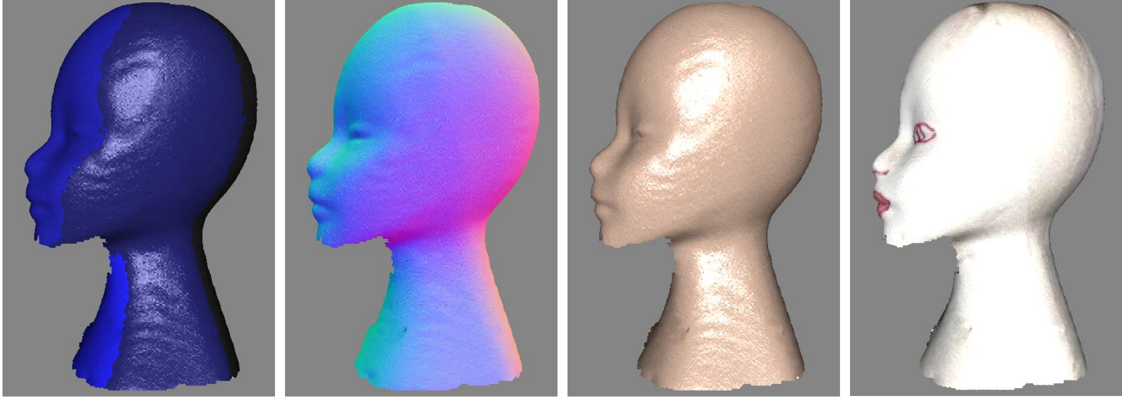


Figure 4.12: The branca model is a laser scanned surface. The charts are shown in different colors.

The main problem with our approach is all the texture transfer between charts. To avoid loss of signal quality, resampling must be done properly in each step. A positive side is that previous works have shown that it is possible to use graphics hardware to accelerate all resampling operations. We discuss the computational aspects of resampling for transferring signals between charts in Chapter 5.

Altogether, editing charts have many advantages:

- Big neighborhoods are well represented;
- Texture space operations have simpler topology;
- Accurate resampling is possible, even leveraging graphics hardware power.

While many possible editing chart parametrizations would be possible, in this work we chose to focus on orthogonal projection editing charts. The main reason is that an orthogonal chart with color and normal attributes is in fact an RGBN image. As was shown in previous chapters, orthogonal projections are a very appropriate representation for normal processing. Additionally, a projection is a very intuitive mapping for user manipulation, in contrast to most low distortion parametrizations. At last, we do not have to store any extra texture coordinates. Any camera position and orientation together with the surface geometry will implicitly define the

Figure 4.13: The face model is a laser scanned human head. The charts are shown in different colors.

parametrization. Editing attributes other that normals in projection charts is currently available in mesostructure processing systems [16, 46]. In fact, these systems usually let the user export the screen image to paint in more specialized image manipulation programs. While this could be done in our system for editing colors, image manipulation programs would be inappropriate for processing normals.

In summary, orthogonal projective editing charts bring the following advantages:

- Easy to implement normal operations;
- Mappings implicitly defined by surface and camera position. No extra texture coordinates required;
- Intuitive for user manipulation.

At the moment, we abstract implementation issues and return to the central problem of this thesis: editing surface normals. All the RGBN operations described in previous chapters can be easily migrated to a textured model by using the orthogonal projection chart. In the next section, we discuss editing signals in domains contained in a single chart.

4.7 Editing Normals

In this section, we assume each orthogonal chart contains colors and normals, thus being a generalization of RGBN images. However, there is an important difference.

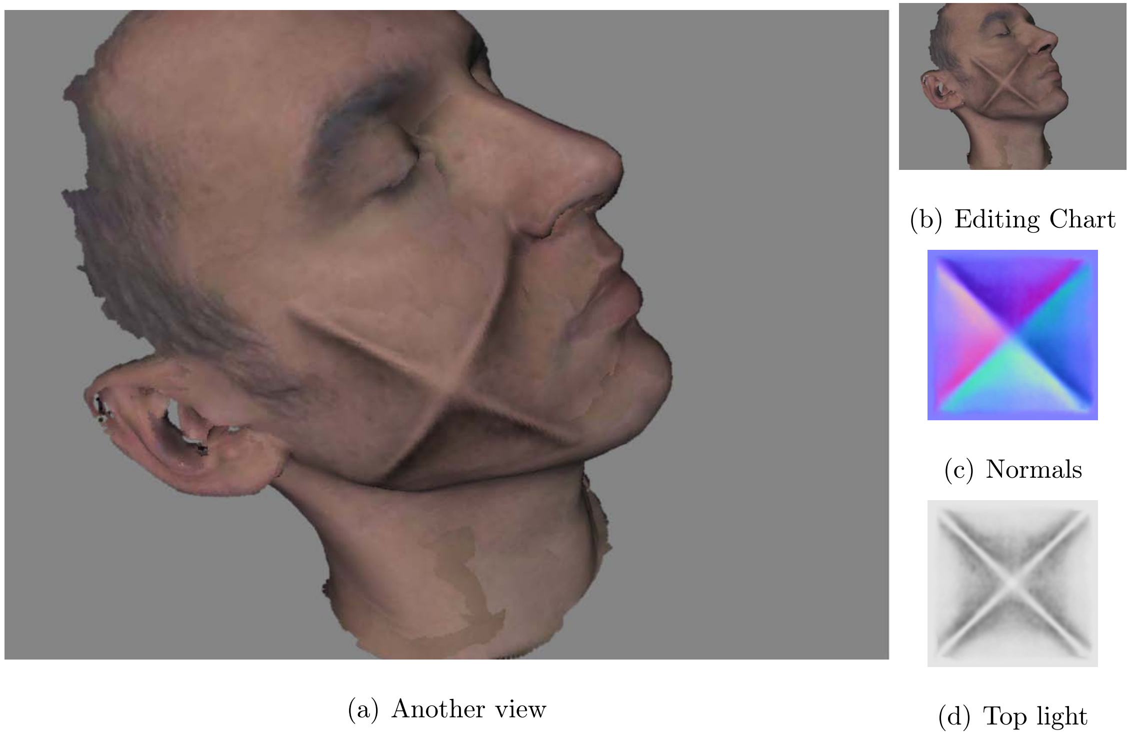
When working with RGBN images we always assumed the up direction to be the coordinate vector direction u_z. In this section, we assume an arbitrary vector u is the up direction of the chart's orthogonal projection mapping. How can we adapt the previous normal operations? All we need is a rotation R that takes u into u_z. First, we rotate all normal vectors to change the coordinate frame. Second, we can perform operations as described before for RGBN images. Finally, we can rotate back the normals to the original coordinate frame. In the next section, we propose an alternative but equivalent process.

All operations in this section are view-dependent operations (Section 4.4). We leave as future work the generation of results with the displacements in the normal direction (Section 4.3).

All results presented in this section are operations on either of two geometric models. Both models were laser scanned and a texture atlas was built for them. The branca model is divided in 4 different charts (Figure 4.12). The face model is divided in 40 charts (Figure Figure 4.13).

4.7.1 Combining Normals

We start by discussing combination of normals, both scalar displacements and view-dependent combination can be applied. The results presented in this section show that the editing chart is an effective way to add details. All facial images in this section are shown illuminated by a light source in the same position as the camera. Figure 4.14 shows an x-shaped scar added to a face both in the chart used for editing and in an alternative view. To generate the model in Figure 4.15, many combination operations were performed each in a different view. The size of the bricks varies from one chart to another to better fit the face. However, the size of the bricks in a single chart also varies, even though all bricks in the normal image were of the same size. This can be seen specially in the forehead. The reason is the editing charts metric is not the object's metric. In other words, due to foreshortening, the spacing between adjacent pixels in screen space is not the same as in the object. In Figure 4.16, words were imprinted on the surface.



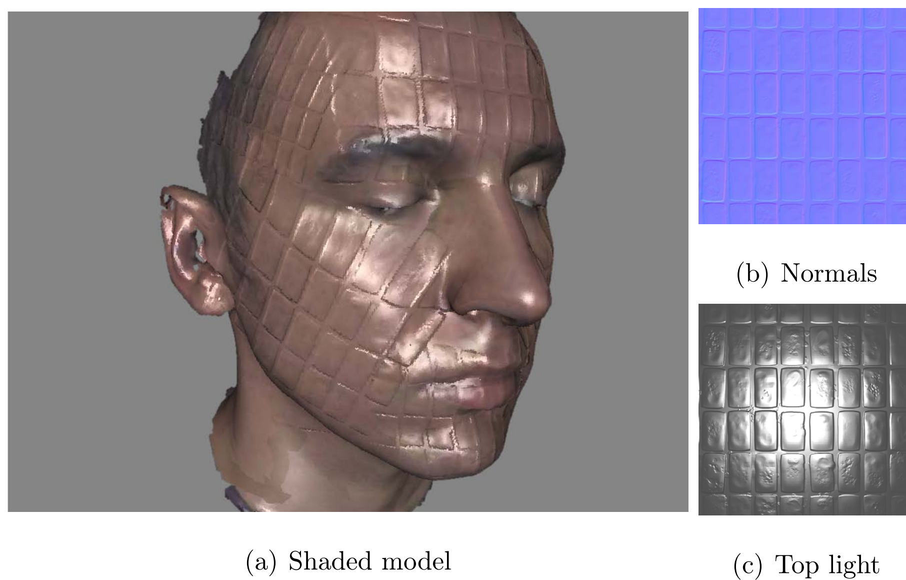
(a) Another view

(b) Editing Chart

(c) Normals

(d) Top light

Figure 4.14: This examples shows an x-shaped scar added to a face both in the chart used for editing and in an alternative view. Models are illuminated by a light source in the same position as the camera.



(a) Shaded model

(b) Normals

(c) Top light

Figure 4.15: To generate this model many combination operations were performed each in a different view.

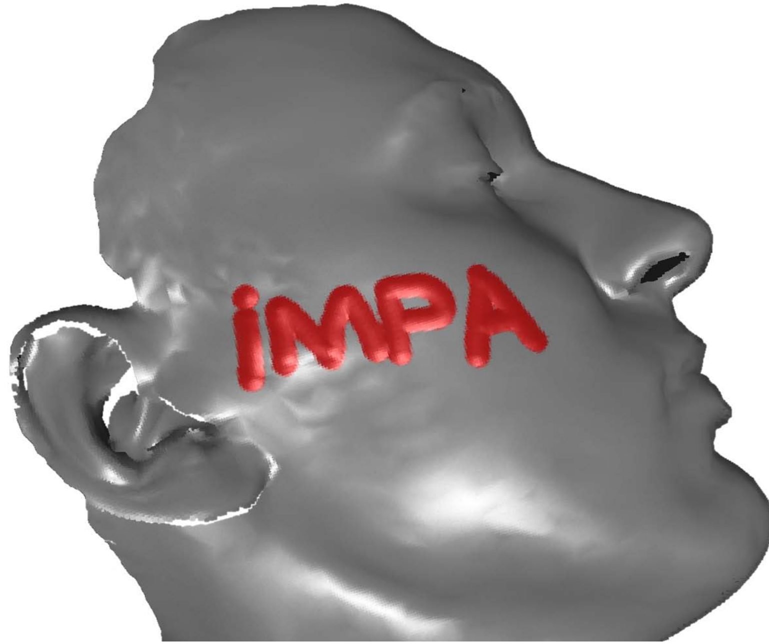


Figure 4.16: Words can be combined with the base surface.

As discussed, in Chapter 2, an important feature of a method for adding details is giving the user control over whether to respect or ignore existing surface detail. In Figure 4.17, we show two images. In the first, the original details of the object were preserved, while in the second they were ignored. As described before, our method can provide this option by combining the new normals with either the original normals or the base normals. The x-shape in this example is very sharp. It also shows how our method can handle sharp features.

There are two cases in which normal combination may create discontinuities. First, when we choose to ignore details (Figure 4.17). The reason is we use the base normals inside the editing region and keep the original normals outside. Second, if the combined normals are not pointing up in the border of the editing region a discontinuity is created. For simplicity our current solution is applying normal blur over the border. More advanced blending operations should be investigated in the future.

(a) Keeping details

(b) Removing details

(c) Normals

(d) Right light

Figure 4.17: An important feature of a method for adding details is giving the user control over whether to respect or ignore existing surface detail.

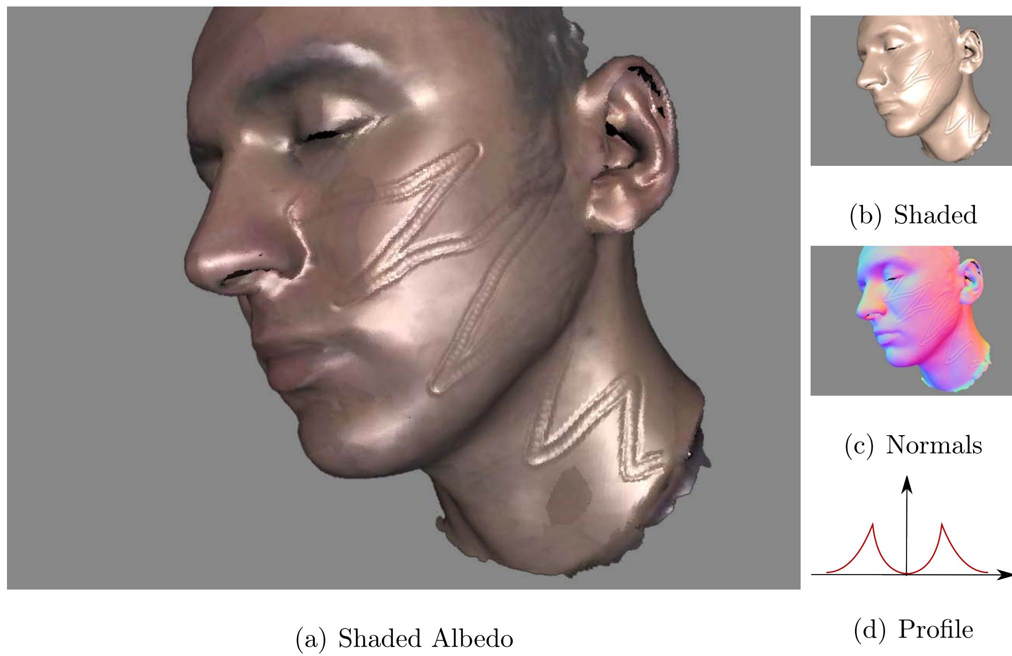
(a) Shaded Albedo

(b) Shaded

(c) Normals

(d) Profile

Figure 4.18: We used a sharp profile to add wrinkles on the forehead of the model.



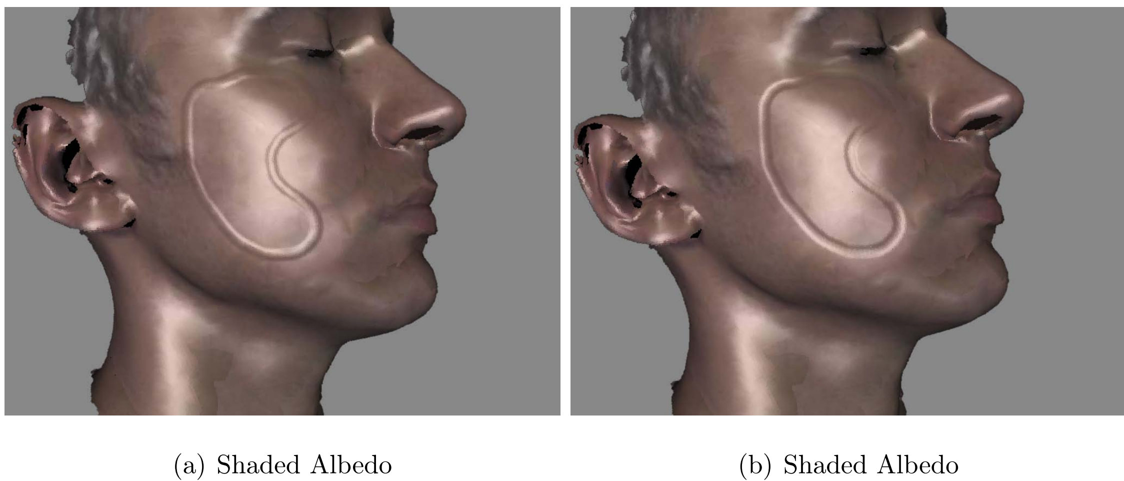
(a) Shaded Albedo

(b) Shaded

(c) Normals

(d) Profile

Figure 4.19: New features were designed by warping a predefined height profile along a user sketch. The resulting normals are combined with the existing normals.



(a) Shaded Albedo

(b) Shaded Albedo

Figure 4.20: Sketching with different profiles can be used to create bumps or to dig the surface.

4.7.2 Sketching Features

As described in Chapter 2, sketching can be a powerful tool for adding new features. It works by warping a predefined height profile along a user sketch. The result is combined with the existing normals exactly like in the previous subsection. In Figure 4.18, we used a sharp profile to add wrinkles on the forehead of the model. In Figures 4.19 and 4.20 a variety of profiles were used.

In designing the features, different lines can be useful. Free-form lines were used to generate the deformations in Figure 4.19. A screen-space straight line was used in Figure 4.18. In the chart used for this deformation a screen-space straight line was approximately an object-space straight line. However, most of the time this will not happen. Instead, we would like to work with surface geodesics. The orthogonal chart would be most useful for calculating this geodesics because, under orthogonal parameterization, the normal vectors can be used to find the surface metric [13].

Additionally, we can raise or lower a region by tracing its border with a pen operator that introduces a level change Figures 4.21. Rarely, inspecting normals through shading can be misleading. Figure 4.22 shows the classical ambiguity present in the shape from shading literature. With the light in this position it is impossible to tell whether this region was raised or lowered. The ambiguity would be broken by moving the camera or the light.

4.7.3 Filtering

In this section, we apply the filtering method from Chapter 2 to the editing chart. A few methods have been proposed for normal filtering both in irregular meshes [63, 64, 65, 66, 21] and in volumetric representations [63]. Compared to irregular mesh based approaches, our parametric normal filtering method is not only more efficient but also allows for kernels of larger support, which would be very difficult to implement over meshes. In addition, any linear filter can be applied to the details in our method.

In [21], Taubin defines a specialized laplacian operator for smoothing normals on

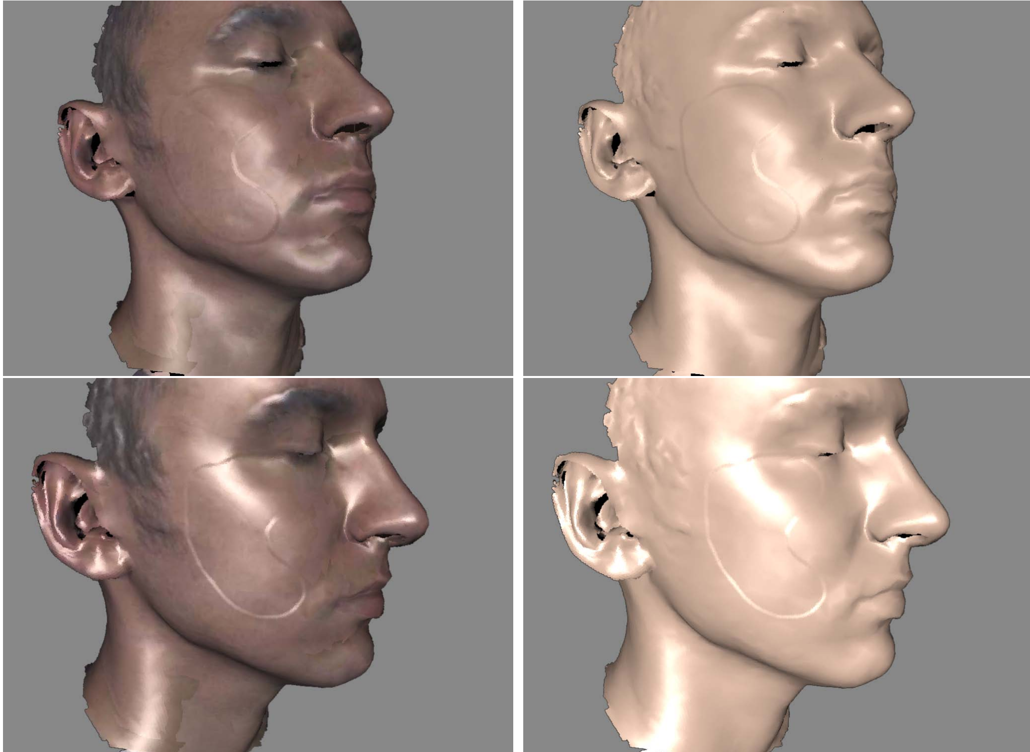


Figure 4.21: In this example, we have used sketching to raise/lower a region by tracing its border with a pen operator that introduces a level change.

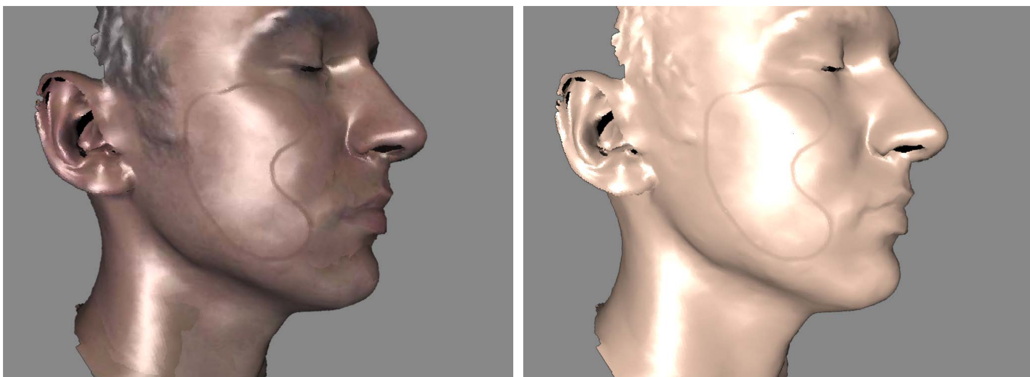


Figure 4.22: Here we show the classical ambiguity present in the shape from shading literature. With the light in this position it is impossible to tell whether this region was raised or lowered. The ambiguity breaks by moving the camera or the light.

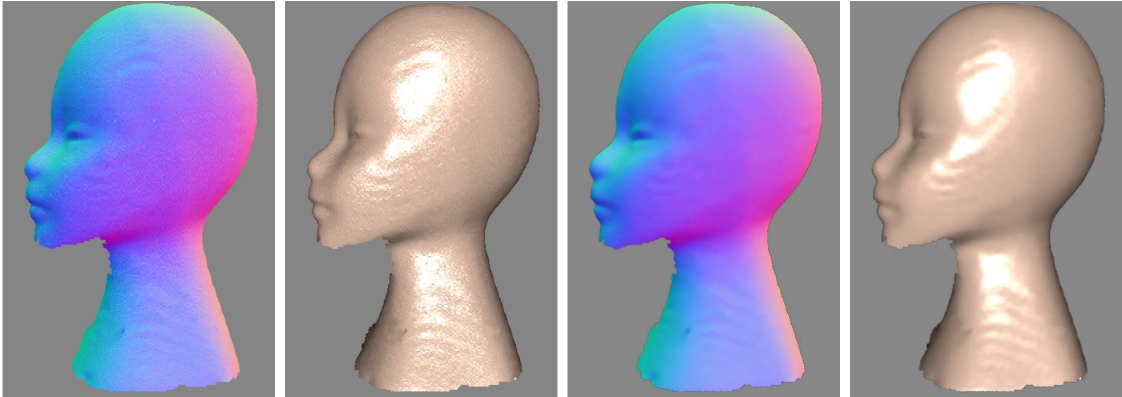


Figure 4.23: Gaussian filtering.

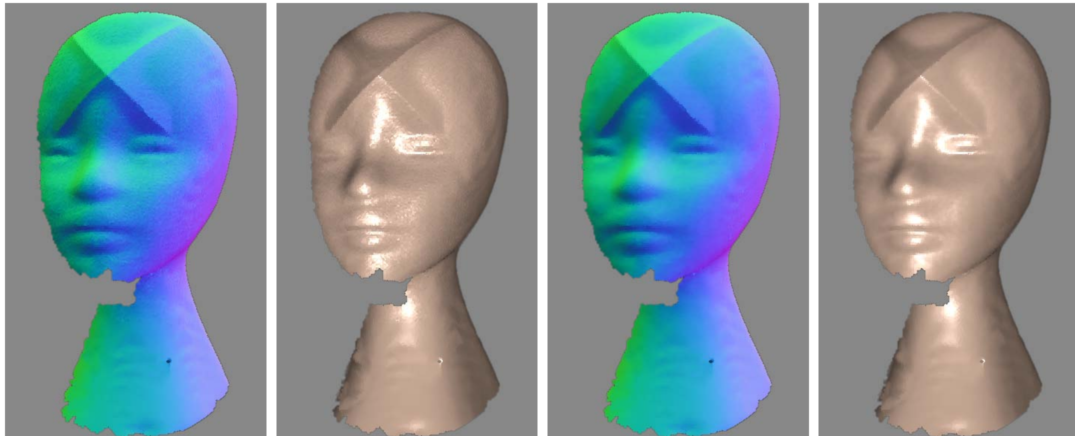


Figure 4.24: Bilateral filtering removes noise while preserving edges.

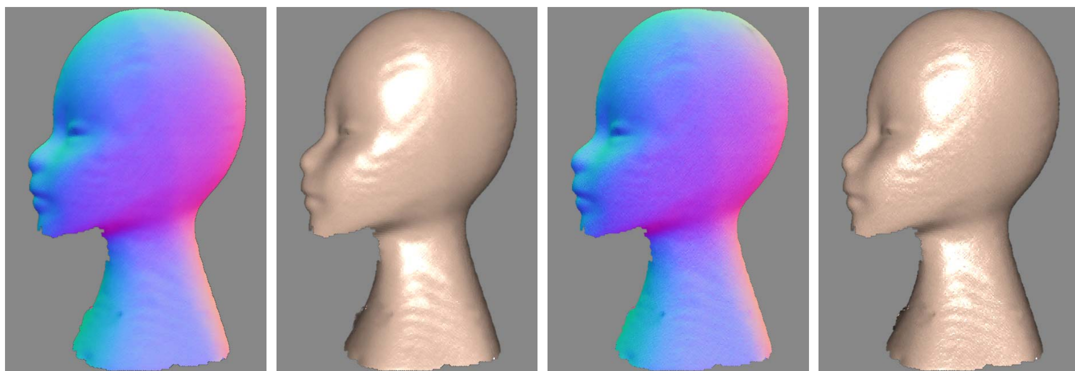


Figure 4.25: High-pass filters can be used for enhancing high frequencies.

a mesh. He then integrates this normal field to obtain filtered positions. In [66], a bilateral filter for normals was adapted to work in an irregular mesh instead of the classical image domain. Our bilateral filter is closer to the classical one, since we work in a parametric domain, as such most bilateral filter optimizations should be possible in our setting.

In addition to smoothing, normal sharpening filters have also been adapted to the irregular mesh context. In [64, 65], the authors apply a high pass filter to mesh normals. Next they adjust the surface to fit these normals using least squares approach.

In [63], the authors use a volumetric representation to define different diffusion operators based in PDEs. In this framework, they first anisotropically smooth a normal field and then reconstruct a level set surface. Compared to volumetric approaches, our intrinsic method is much more efficient regarding storage, as such models of much higher resolution can be processed.

In [23], the authors formalize normal map smoothing filtering using convolution between normal distribution functions and the BRDF. However, at each point their filtering result is not a normal, instead it is a normal distribution. Similar to us, they use textures to represent the signals. However, they use mip-mapping methods for filtering which does not represent well neighbourhoods near chart borders.

As we have shown in Chapter 2, we can apply any linear filter to normals. In addition, with the separation of base surface and details proposed in this chapter, we can restrict our filters to only operate on details, preserving the base surface.

All results in this section apply view-dependent filters in a single chart. But all methods equally apply to normal displacement filtering. In Figure 4.23, Gaussian filtering was applied to smooth the normals. The problem with Gaussian filtering is that edges are blurred. In Figure 4.24, we have used the RGBN bilateral filter previously proposed in [13] on the editing chart. It preserves the edges of the x-shape. Additionally, in Figure 4.25, we use an unsharp mask filter to enhance detail. The editing chart lets us extend any RGBN filter to normal maps in a texture atlas.

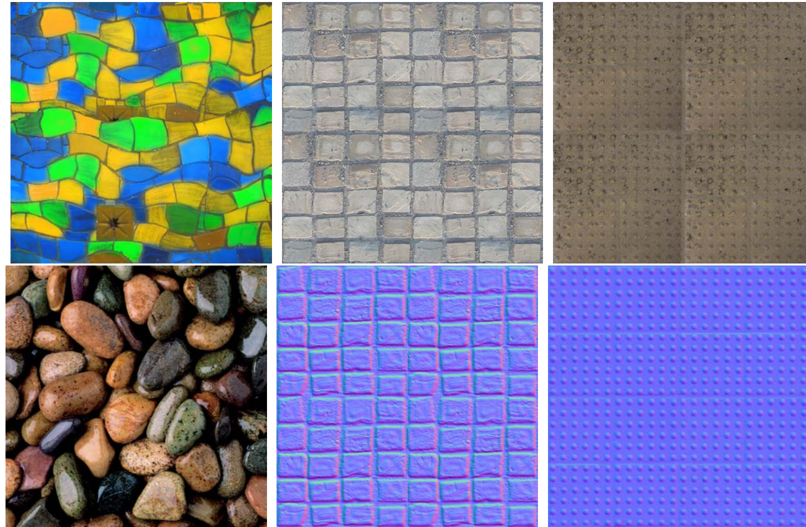
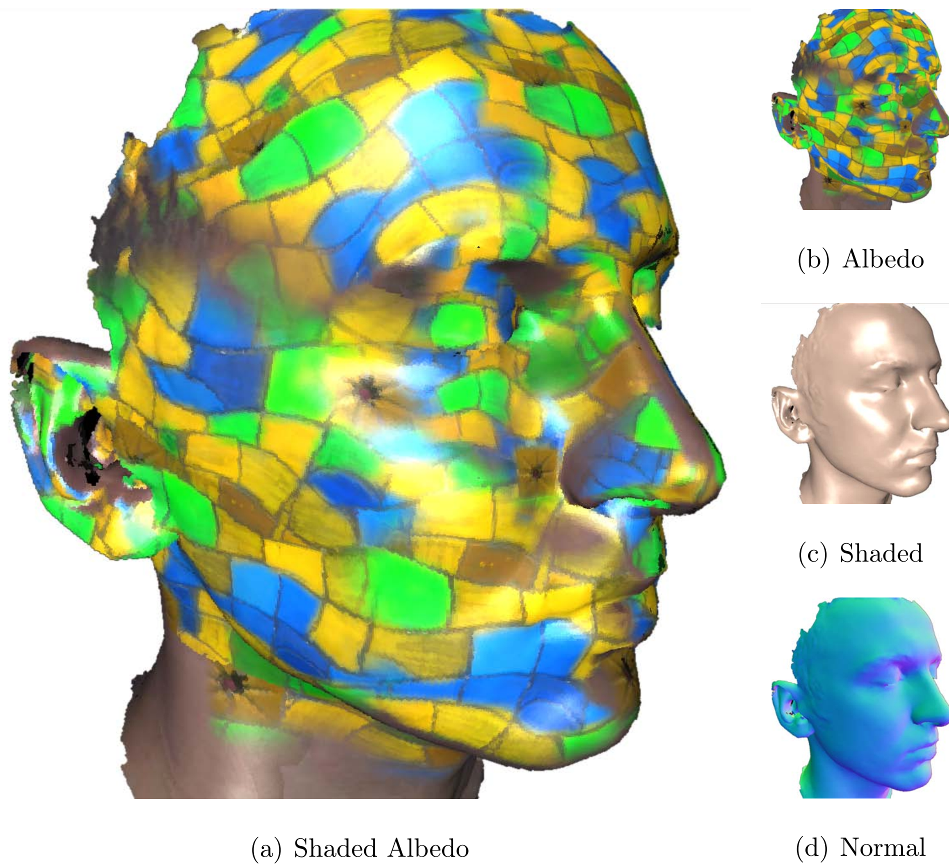


Figure 4.26: RGBN samples used in examples.

4.7.4 Editing Multiple Charts

In this section, we use multiple viewpoints to perform more complex attribute editing. We used blending to avoid creating seams between editing charts. We detail the blending procedure in the next chapter. In Figure 4.27, we used multiple views to apply a color pattern. In Figure 4.28, we first applied a color pattern, next we used color filtering in half of the face. Two charts were used to filter just the left half of the face. We also show examples where a full RGBN image was applied to the face model. In Figure 4.29, bricks were added to the face. We have varied the scale of the bricks manually to better match geometry. Figure 4.29-d shows the new geometry without the influence of albedo. In addition, in Figure 4.30 we added multiple bumps creating a reptile appearance. All patterns used in these examples are shown in Figure 4.26.



(b) Albedo

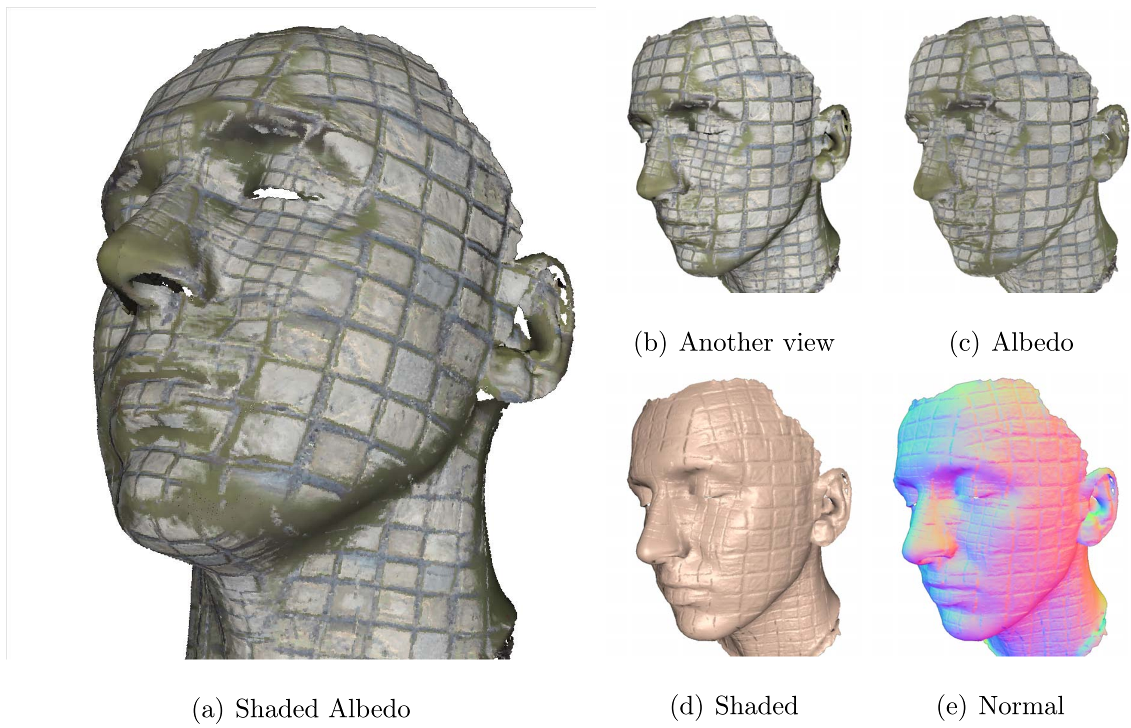
(c) Shaded

(a) Shaded Albedo

(d) Normal

Figure 4.27: Color pattern applied to face using multiple viewpoints and blending.

Figure 4.28: In this example, first we applied a color pattern, next we used color filtering in half of the face. Two charts were used to filter just the left half of the face.



(a) Shaded Albedo (b) Another view (c) Albedo (d) Shaded (e) Normal

Figure 4.29: We added an RGBN image containing bricks to face model. We have varied the scale of the bricks manually to better match geometry.

Figure 4.30: Face with multiple bumps added, creating a reptile appearance.

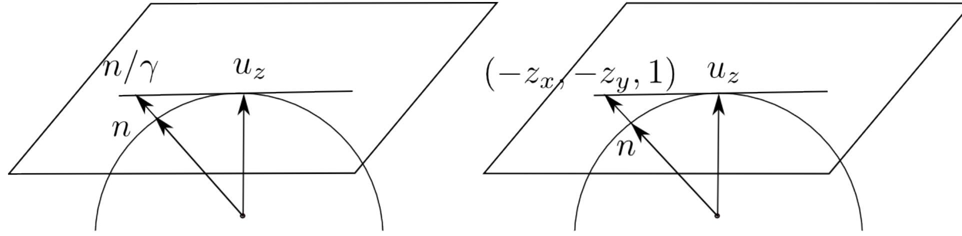


Figure 4.31: Geometric representation of the conversion between normals and gradients. This sphere is the Gaussian sphere, i.e., the unit normal sphere which contains the image of the surface's normal map.

4.8 Tangent Plane Method

In Chapter 2, we showed two operations to convert between normals and gradients. These conversions were used to define more powerful operations like normal filtering and linear combination. In the previous section, we have seen how these conversions can be extended to surfaces using rotations to change coordinates systems taking normals from the editing chart to the canonical orientation. In short, we first apply a change of basis, followed by the conversion to gradient. In the gradient domain we can manipulate the geometry. Having completed the operation, we convert from gradients to normals and rotate them back to the original basis. In this section, we present generalized conversions that already incorporate the change of coordinate frame. Hence, we avoid all rotations. The resulting method, which we call the tangent plane method, is simpler to implement and it has fewer primitive operations. In its present formulation, it only applies to view-dependent processing (Section 4.4). We leave as future work its extension to scalar displacement processing (Section 4.3).

As we have seen in Chapter 2, given a parametrization $\psi(x, y) = (x, y, z(x, y))$, we use the following formula for conversion between gradients (z_x, z_y) and normals n:

$$n(x, y) = \frac{\psi_x \times \psi_y}{|\psi_x \times \psi_y|} = \frac{(-z_x, -z_y, 1)}{\sqrt{z_x^2 + z_y^2 + 1}}$$

To convert from a normal $n = (\alpha, \beta, \gamma)$ to a gradient we use: $-\alpha/\gamma = z_x, -\beta/\gamma = z_y$, as such $n/\gamma = (\alpha, \beta, \gamma)/\gamma = (-z_x, -z_y, 1)$.

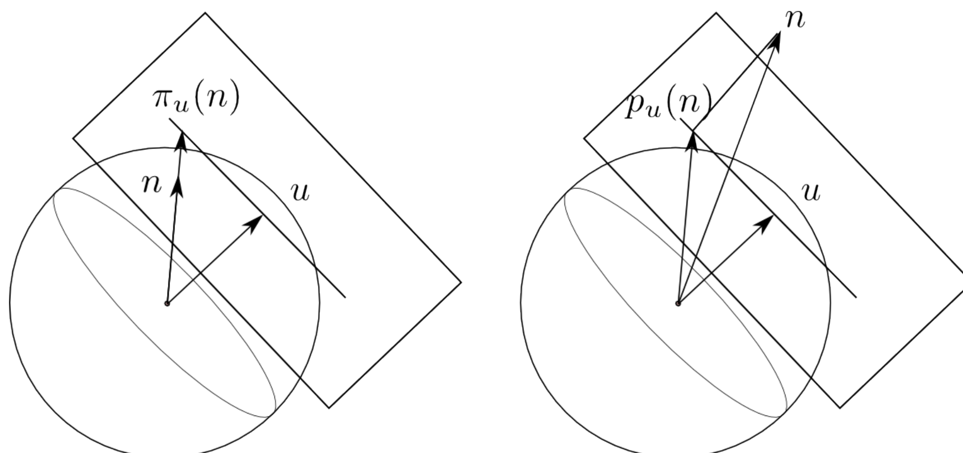


Figure 4.32: The operations $\pi_u(n)$ and $p_u(n)$ are used to generalize the conversion between gradients and normals to any viewing direction u.

Figure 4.31 presents a geometric interpretation of these conversions. It shows how the conversion to gradient consists in scaling the normal vector until the z component is equal to one. The main change required to generalize these formulas is replacing the default up direction u_z by a general view direction u. Therefore, its generalization is scaling the normal until the component in the u direction is one:

$$\pi_u(n) = \frac{n}{\langle u, n \rangle}$$

It is easy to verify that the u component of this vector is indeed one, as can be seen in Figure 4.32. Next, any linear combination of these projected vectors can be performed having the same effect of combining gradients. Similar to the case of RGBN images, the combination of these vectors can yield vectors whose component in the u direction is not one. Since, this component does not carry any information, it is simply restored to one with a projection:

$$p_u(n) = n + u - \langle n, u \rangle u$$

It is once again easy to verify that the u component of this vector is one $\langle p_u(n), u \rangle = \langle n, u \rangle + \langle u, u \rangle - \langle n, u \rangle \langle u, u \rangle = 1$. Finally, we can apply the normalization operator U to this vector to produce the final unit normal $U(p_u(n))$.

If we set $u = u_z$, the two above operations are simply the original RGBN image ones. Note that $\pi_u(n)$ cannot be performed in case u and n are orthogonal. In this case, another editing chart must be chosen to edit this point.

To prove that the tangent plane method produces the same results that the change of coordinate frames methods involves some manual calculations. As an example, we analyze here the case of filtering operations. Each filtered point is actually a linear combination of many gradients $\pi_{u_z}(R(n_i))$ using weights from the filter's kernel. For simplicity, we assume our linear filter result n_c is only a linear combination of two normals n_a, n_b. This filter in the change of basis method can be written as:

$$n_c = R^{-1} \circ U \circ p_{u_z}(\pi_{u_z}(R(n_a)) + \pi_{u_z}(R(n_b)))$$

In this formula, R is a change of basis given by a rotation such that $R(u) = u_z$. In fact, it is the only linear operator above. However, since the other operations are scalings or projections, they all share some interesting commuting properties which will be used to simplify the above formula.

Proposition 1. $R \circ U = U \circ R$

Proof. $R \circ U(n) = R(\frac{n}{\sqrt{\langle n,n \rangle}}) = \frac{R(n)}{\sqrt{\langle n,n \rangle}} = \frac{R(n)}{\sqrt{\langle R(n),R(n) \rangle}} = U \circ R(n)$, where we have used that R is an orthogonal linear transformation and thus preserves dot products.

Proposition 2. $\pi_{R(u)} \circ R = R \circ \pi_u$

Proof. $R \circ \pi_u(n) = R(\frac{n}{\langle u,n \rangle}) = \frac{R(n)}{\langle u,n \rangle} = \frac{R(n)}{\langle R(u),R(n) \rangle} = \pi_{R(u)} \circ R(n)$

Proposition 3. $p_{R(u)} \circ R = R \circ p_u$

Proof. $R \circ p_u(n) = R(n + u - \langle u, n \rangle u) = R(n) + R(u) - \langle R(u), R(n) \rangle R(u) = p_{R(u)} \circ R(n)$

Using these three properties we can simplify the equation for the change of coordinate method:

$$n_c = R^{-1} \circ U \circ p_{u_z}(\pi_{u_z}(R(n_a)) + \pi_{u_z}(R(n_b)))$$

Using propositions 1 and 2:

$$= U \circ R^{-1} \circ p_{u_z}(R(\pi_u(n_a)) + R(\pi_u(n_b))$$

By the linearity of R:

$$= U \circ R^{-1} \circ p_{u_z} \circ R(\pi_u(n_a) + \pi_u(n_b))$$

Using proposition 3:

$$= U \circ R^{-1} \circ R \circ p_u(\pi_u(n_a) + \pi_u(n_b))$$

$$= U \circ p_u(\pi_u(n_a) + \pi_u(n_b))$$

The last equation is exactly the tangent plane method described above, completing the proof that both methods are equivalent. It involves no change of coordinate systems.

4.9 Conclusion

In this chapter, we have presented a solution for editing surface normal vectors. We use a texture atlas to store detailed normal vectors. We use additional temporary ortographic charts for editing. We have shown how many normal operations are easily defined in this chart, including all RGBN image operations. Up to now, the solution described satisfies some of the requirements posed in the introduction:

- Support to operations on details. In particular, we can add new details but respect the existing ones;
- Ability to work in selective frequency bands, respecting the base surface/details separation;
- Ability to handle two dimensional surfaces of complex topology;
- Intuitive editing;
- Transparent representation of data.

The following requirements depend on computational aspects of the solution which are presented in the next chapter:

- Interactive editing;
- Workspace with realtime lighting feedback (WYSIWYG);
- Representation of very high resolution details.

Chapter 5

Chart Transfer

In this chapter, we present a computational solution for editing surface signals with the editing chart. As was discussed in the previous chapter, we use a texture atlas to store attributes but use an additional temporary editing chart for each operation. The two main required operations are building editing charts and updating storage charts. The focus of this chapter is the transferring of information between these charts.

From a mathematical point of view, information transfer operations between charts are symmetric operations. It should not matter whether we are going from storage to editing or the reverse, after all, they are all charts. From a signal processing point of view, they are all images and both transfers are indeed resampling operations. Even from a graphics point of view, we could consider transfers as rendering operations. All these views share some truth and these aspects should be remembered while reading this chapter. However, representation and implementation aspects lead us to different analyses and solutions for each operation.

All data transfers should be made efficient. The reason is they are a key component of an interactive editing system. This chapter discusses some of the aspects of system design, focusing on how they relate to chart transfers. For example, in addition to building an editing chart through rendering, our system must also render real-time lighting of very detailed models for user feedback.

	Build editing	**Update storage**	**Representation**
Hanrahan '90	Micropolygons	Id-buffer	Parametric
Hart '04	Textured triangles	Textured triangles	Atlas
Pointshop 3D '02	Point-based rendering	Id-buffer or new samples	Point cloud
Our triangle method	Textured triangles	Id-buffer	Atlas
Our point method	Point-based rendering	Id-buffer	Atlas

Table 5.1: Comparison of different texture editing solutions.

5.1 Related Work

An important subclass of texture editing operations is painting. By painting we mean the final color of a texel will only depend on its original color and the color of an associated brush point. In particular, filtering does not fit this description. Early works in painting did not build an editing chart, they only used the update map to resample the brush in storage space. However, they all used some method for screen rendering which we are going to discuss under the chart building label. Different previous texture editing solutions are presented in Table 5.1 together with the two methods proposed in this work.

Hanrahan et al. [55] were the first to propose texture painting on surfaces. They worked with single parametrization models and they use micropolygons, i.e., texel-sized polygons, to render texture and geometry. During rendering they build an *id-buffer*, an off-screen buffer holding for each pixel the uv coordinates of the texel it came from (Figure 5.1). This buffer explicitly keeps the mapping back to storage. Using this buffer, a brush is resampled to the texture where the painting operation actually happens. Additionally, Hanrahan et al. [55] point out that painting can be optimized by selective updates, i.e., only redrawing the modified parts of the object. This is a very desirable characteristic of editing systems allowing large performance

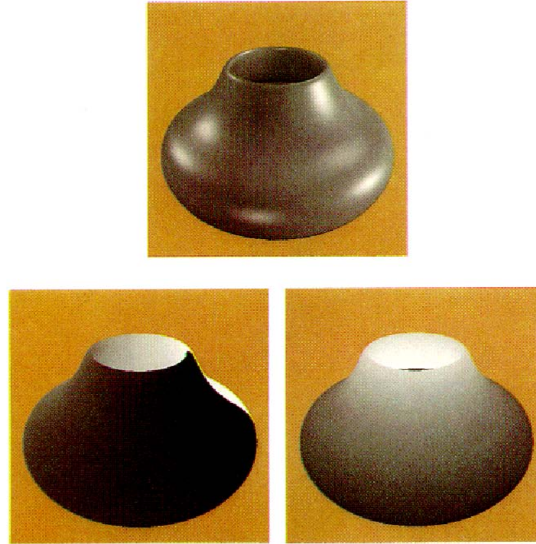


Figure 5.1: An id-buffer can store for each pixel the uv coordinates of the texel that influenced it. In this way, this buffer encodes the mapping necessary for the storage update operation. The top image shows the image as seen by the user. The bottom images show the u and v coordinates in grayscale coding. Image taken from [55].

gains. However, their solution has some drawbacks. Besides not doing general editing operations, their objects are composed of a single parametrization, which brings strong topological limitations. In addition, their method is now outdated by the changes in graphics hardware over the last 20 years.

Two new technologies influenced the painting system design. The first significant change is the use of texture mapping for color rendering in building the editing chart. Although micropolygons are still a common approach to render geometry textures like normal or displacement maps, they were replaced by texture mapping for color rendering. The second change was the development of graphics hardware capable of rendering to texture, i.e., rendering to texture buffers instead of the framebuffer. Render to texture allows efficient solutions for the storage update operation.

Both these technologies are present in the work of Carr et al. [56] who proposed a design making heavy use of programmable hardware. In addition, their solution requires shadow mapping capabilities to correctly solve the visibility problem in the

storage update. They overcome the previous topological limitations by storing colors in a texture atlas, representation similar to ours. Their method could be adopted for normal editing, but pixel shaders would have to be used to render normals.

Meanwhile, the graphics community introduced another technique for representation and rendering of large detailed models: point-based graphics [67]. Influenced by these developements, Pointshop 3D [57] presented an attribute editing solution which, unlike previous solutions, does not use storage charts. All attributes are represented as a point cloud. Like us, they use an editing chart for operation. But different from our solution, their editing chart is not an orthogonal projection. They build a custom parametrization in the region of interest, mapping the points to a chart where operations are performed. Resampling for both screen and editing charts is done with point-based rendering [68] methods. To update their point cloud storage, they either use an id-buffer to update the points or create a new sample point for each edited texel to avoid resampling. In this work, we present a point-based rendering solution for chart transfer. Unlike Pointshop, we store attributes in textures and use point-based methods only to render.

In regards to *building* the editing chart, we can classify methods into those that use textured-triangle rendering and those that use point-based rendering (Table 5.2). Both methods are sufficiently efficient even on low-end hardware for the interactive design we propose. However each method has its advantages. First, while triangle methods require pixel shaders to support normal mapping, point-based solutions naturally render one normal per point. Second, the triangle method only allows large selective updates, if only a single pixel changes the entire triangle has to be re-rendered. Finally, a major disadvantage of point methods is losing object topology. Since each point is rendered separately, there is no connectivity information. This impacts attribute reconstruction in the new domain.

In regards to *updating* storage charts, we can classify methods in many ways. First, there are methods that use render to texture hardware features or cpu-based methods. To the best of the authors knowledge, the work by Carr et al. [56] is the only one using render to texture in a painting pipeline. However, it can be used

	Textured triangles	Point-based
Normal Mapping	Pixel Shader	Direct
Speed	Fast with Display lists	Fast
Selective Updates	Triangle size	Pixel size
Topology	Preserved	Lost - infer with depth

Table 5.2: Comparison of different solutions for building editing charts.

in any architecture. One advantage of using hardware for rendering is the natural antialising it provides. In contrast, antialising has to be dealt with when using direct approaches. Second, methods can be classified according to the rendering primitive in either triangle-based methods or point-based methods. Third, there are methods that use selective updates or simply re-render the entire object. We advocate the use of selective updates as a mean to use resources wisely. Nevertheless, some attribute editing methods do not use it because selective updates are complex to use when deforming geometry. When position changes, it is hard to know where each new texel is going without re-rendering the whole model. Hanrahan [55] did not face these problems because he only edited color not geometry. Similarly, we do not have these problems because we only change geometry encoded in the normal map in addition to color. Therefore, each texel is still going to influence the same screen pixels. Either using triangles or points as primitives, selective updates require knowing the primitives contained in a subset of the editing chart. This can be solved with an id-buffer or with picking.

	Render to Texture	Selective Updates	Primitive
Hanrahan	No	Yes	Point
Hart	Yes	No	Triangle
Pointshop 3D	?	Yes	Point
Our triangle method	No	Yes	Point
Our point method	No	Yes	Point

Table 5.3: Comparison of different storage update solutions.

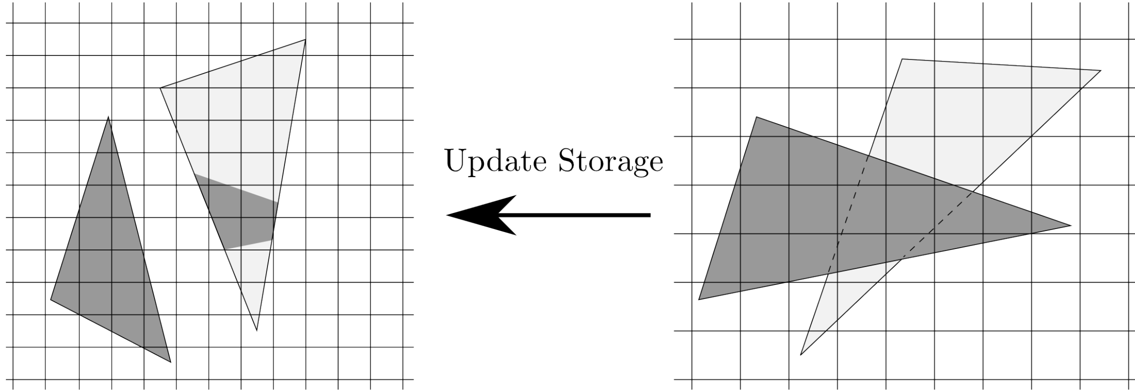


Figure 5.2: During a render to texture operation with texture mapped triangles, pixels that were not visible from the camera may be wrongly changed.

The building and updating transfer operations must deal carefully with the visibility problem. The problem is that during the storage update only texels that were visible from the camera should change. It is rather easy to solve visibility for building the editing chart using the z-buffer. On the contrary, it is complicated to handle it correctly for updating. This asymmetry comes from hardware which is usually dedicated to one direction, i.e., dedicated to painting the framebuffer. Rendering triangles to storage using the editing chart as texture may change pixels that were not visible (Figure 5.2). Carr [56] proposed to solve this problem using a shadow buffer, feature only available as an OpenGL extension. On the other hand, the id-buffer method, which we follow in our work, provides a simple solution. Since operations are done on a per-pixel, rather than per-triangle basis, only pixels that show on the screen will affect storage.

Even though color textures and geometry are now decoupled using hardware texture mapping, micropolygons are still a common approach to render geometry textures such as normal or displacement maps.

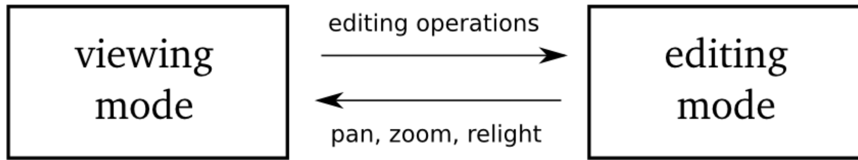


Figure 5.3: Editing modes.

5.2 Method

In this section, we propose an editing system design appropriate for normal editing that implements the editing chart method. Our method uses an id-buffer to update storage. In addition, for building the editing chart, we present two different methods: the triangle method and the point method.

Before going into the details of each operation, we describe the editing loop. More specifically, the case of interactive operations, since non-interactive ones would be much simpler to implement. In editing systems, besides editing, the user can change lighting or change the view, including zooming, panning or rotating. An important remark is that the user never does these operations simultaneously. Therefore, our system works in these natural working modes: editing and viewing (Figure 5.3). On the one hand, in editing mode, we assume lighting and viewing conditions do not change. Although not stated in the last subsection, these hypothesis allow selective updates to work. We use selective updates to optimize rendering. This allows us to use resources wisely and not require real-time rendering for models that are being edited. We avoid costly data transfer operations between CPU and GPU. On the other hand, in the viewing mode, we assume the model does not change. With this restriction, the data can be entirely transfered and stored in the GPU at once, for example using display lists. Consequently, we avoid an overload of CPU/GPU data transfer and can render the object in real-time.

Knowing when to change modes is rather easy. If the system is idle or during a painting operation, we leave the system in editing mode. When the user finishes a stroke, all information is transfered to the GPU. However, the system only goes

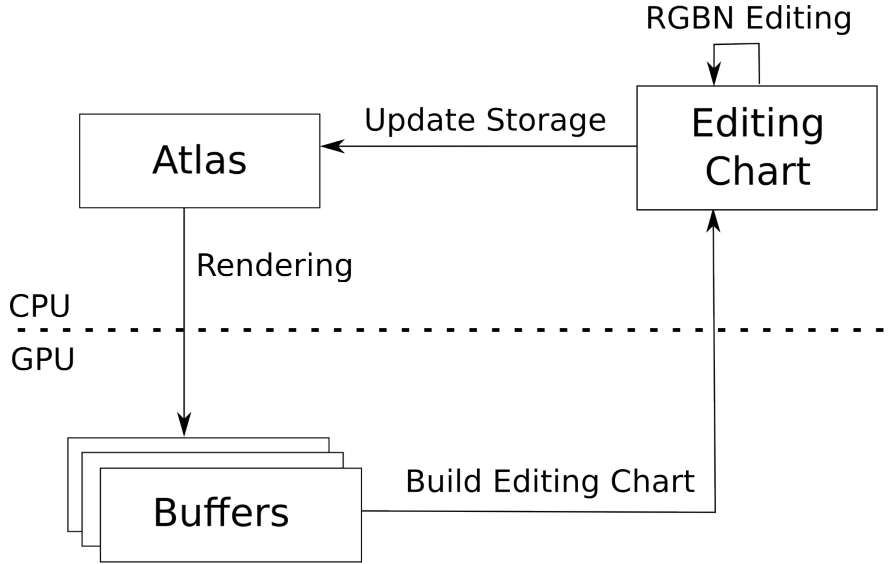


Figure 5.4: Editing loop.

to viewing mode if the user starts any of pan, zoom, rotate or relight operations. In viewing mode, real-time visualization is possible. When the mouse is released, finishing the viewing operation, the system goes back to editing mode, closing the cycle.

Figure 5.4 depicts the implementation of the chart transferring mechanism proposed. Both the storage atlas and the editing chart are resident in main memory during operations. To build the editing chart we first use OpenGL to render image buffers which are then read back from the GPU to the main memory. We render three buffers: albedo, normal and id-buffers. Similarly, other attribute buffers could also be used. In addition, we render an illuminated version of the object, in fact, the only image the user can see. Altogether, there are four rendering passes. In short, rendering and read back operations implement the editing chart building operation previously described.

With the editing chart at hand, we can use the editing operations from the previous chapter. After editing the storage update operation must occur. For non-interactive operations, e.g., bilateral filtering, we only update the screen at the end of the operation. For interactive operations, e.g painting, we use selective updates

to provide the user with instant feedback. If editing ends and the user changes the view, the system goes to viewing mode. All the model is transfered to the GPU using display lists and is kept there for real-time rendering. In viewing mode, no editing occurs so only single-pass rendering is used. The more costly four pass operations are only seldom performed, only when the system changes to editing mode. In our architecture all operations are performed using standard OpenGL functionality. Performance has shown adequate for interactive editing. However, editing even higher resolution models should be possible with the implementation of the entire editing loop in the GPU.

In the next sections, we provide the details of our method, including two different solutions for building the editing chart and an id-buffer based solution to update storage charts.

5.3 Chart Transfer

As stated in section 5.1, there are multiple choices in the design of a solution to edit textures. In this section, we present two different methods: using textured triangles and using points. Both solutions build an id-buffer to support the subsequent storage update operation (Section 5.3.3).

5.3.1 Triangle-based Method

Our texture-mapped triangle solution to chart editing as a whole can be considered a hybrid of the approaches of Hanrahan [55] and Carr [56]. The reason is we use texture mapping to build the editing chart, but we use an id-buffer to update storage. Compared to Carr, by using the id-buffer, we have pixel level visibility and have no need of shadow mapping techniques. Additionally, the id-buffer supports direct access to texels, thus avoiding the need for render to texture hardware features.

Rendering texture mapped triangles is a straightforward technique. For both albedo and normals, bilinear or trilinear interpolation can be used. However, caution must be taken when building the id-buffer (Figure 5.5) using color interpolation.

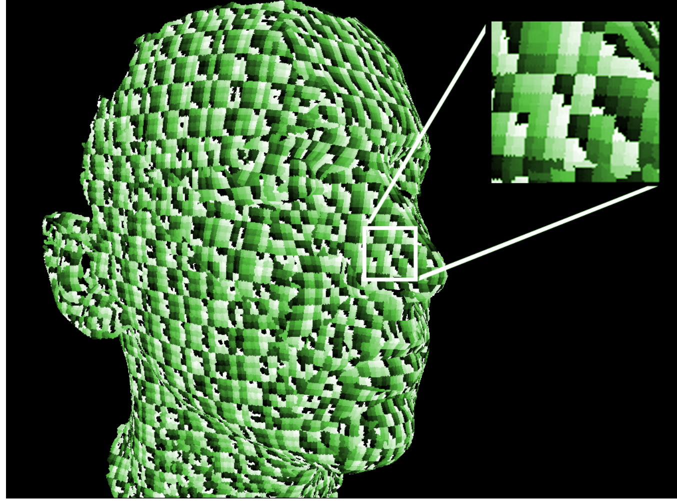
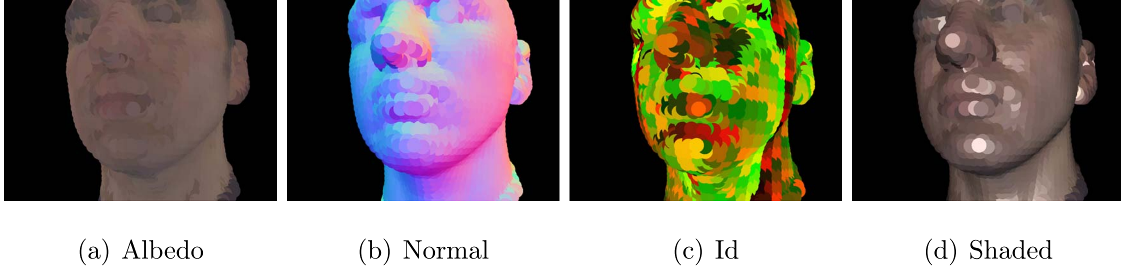


Figure 5.5: Id buffer built with triangle rendering.

Hanrahan built his id-buffer without using texture mapping, therefore he did not face these issues.

First, we could use a single color channel to encode the u or v coordinates, but that would only give us 256 possible values, which is far too small for a texture. For this reason we use two color channels for each coordinate. For example, the u coordinate is encoded in 16 bits, the red channel plus the green channel, analogously for v using blue and alpha channels. This gives us a maximum texture size of 64k in both width and height. The simple solution to encode u is sending its first 8 bits to the red channel and the last 8 bits to the green one. We can think of this mapping as coding function $c : R \rightarrow R^2, c(u) = (r(u), g(u))$. While this function is a bijection, it is not continuous. This is easy to see, since $r(u) = u \mod 256$, which is not continuous in large intervals, e.g, r changes abruptly when u goes from 255 to 256.

Continuity is a big problem when using interpolation during triangle rasterization. The coded uv coordinates from the vertices are linearly interpolated using the actual uv coordinates. For this reason, the coded color must be a continuous function of u, v. In fact, it is simple to fix the above problem with c. While the simple description above can be interpreted as a raster scan of r, g space, what we actually



(a) Albedo (b) Normal (c) Id (d) Shaded

Figure 5.6: Different buffers rendered with point-based rendering. A low resolution model with big point sizes is used here for illustration purposes only.

need is a serpentine scan. It should alternate between moving forward and backwards. Depending on whether g is odd or even, r will be equal to either $u \ mod \ 256$ or $255 - (u \ mod \ 256)$. Serpentine scan provides a continuous parameterized path in r, g space.

Editing colors with this texture-mapped solution would work perfectly. However, editing normals brings additional requirements. We want to provide the user with instant feedback in the form of a lighted model. This triangle solution can provide interactive feedback using normal mapping. However, normal mapping requires GPU programming. In the next subsection, we discuss a point-based solution that provides feedback on changed normals using point-based graphics.

5.3.2 Point-based Method

In addition to the triangle rendering solution, we also present a point-based solution to the editing chart building problem. It should be clear that we do not switch to a point-based representation of the model. Instead, we use the texture atlas representation and only use point splatting [68] to render these textures creating the editing chart (Figure 5.6). One big advantage of point rendering in our normal editing system is the automatic support of normal mapping.

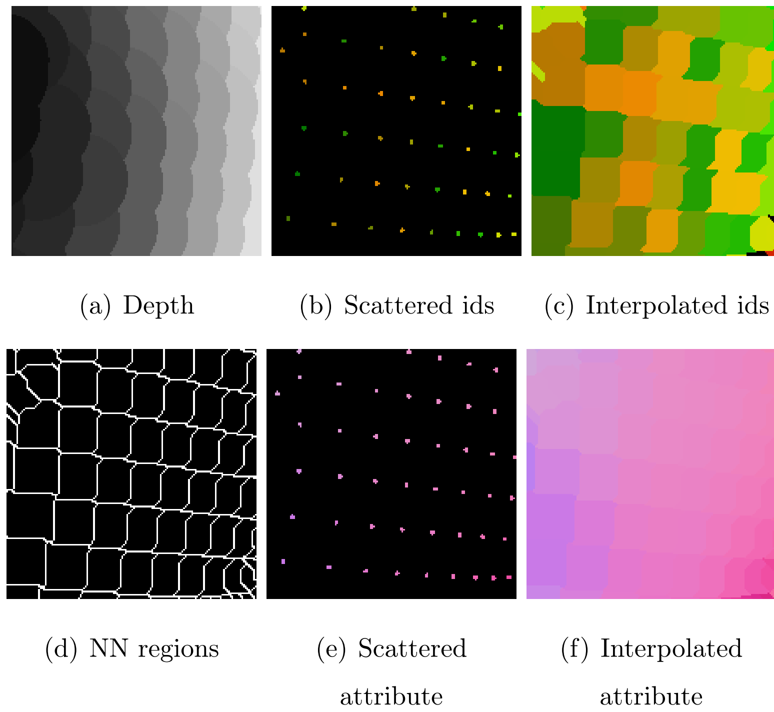
For this purpose, we create one point sample for each texel. Since these texels already contain color and normal information in high resolution, we must only find the position of each texel. Based on our separation of the different levels of detail,

the base surface positions are usually stored in a different representation and must be converted. In our implementation, we had original surface positions in a triangle mesh. Therefore we rendered the smooth position on the storage charts, defining each texel's position in a manner similar to the geometry images approach [3]. With all the information sampled per point, we use point-based rendering methods to build the editing chart and to render the screen image for user feedback. All examples in this work, use a projective atlas [38] as the storage atlas. However, any other atlas could have been used. At the end of this section, we discuss justify our choice.

After building 3D point samples, the next step is rendering. One of the most important decisions of this operation is determining the screen size of the splat. If the splat is too small, the final surface image will contain many holes. In addition, points that do not belong to the visible surface may show through theses cracks. In contrast, if the splat size is too big, they could overcrowd and some texels would simply not influence the editing chart. While losing a few points may not be a great concern for most point-based rendering applications, it is indeed a concern to our application. The reason is we are using point rendering to build a chart. As such, it must be a bijection, no texels can be lost. For example, if a given texel is lost in rendering and the editing operation is filling the texture with a new color, the lost texel will simply keep its original color creating an artifact.

In our work, we use two different sizes (Figure 5.7). First, we use big splats to render the depth-buffer. Using big splats the resulting depth buffer does not contain any holes. It is thus used to solve visibility problems in rendering the next buffers. Second, we use small splats to render the id-buffer and attributes. This way we guarantee that no ids are lost and the map is a bijection. An interpolation step must follow to recover attributes and ids from these scattered samples.

Nevertheless, using a traditional depth buffer visibility test fails when working with big splats. Big splats leave no holes in the screen and effectively eliminate any points which should not be visible. However, the visibility test may eliminate even visible points. The reason is a splat may get discarded when a neighbor splat fills the depth buffer with a smaller z value. The solution is inserting a small tolerance



(a) Depth (b) Scattered ids (c) Interpolated ids

(d) NN regions (e) Scattered attribute (f) Interpolated attribute

Figure 5.7: A big splat size is used to recover the z-buffer (a). Small splats (b,e) followed by a nearest neighbor interpolation (d) to recover ids and attributes (c,f).

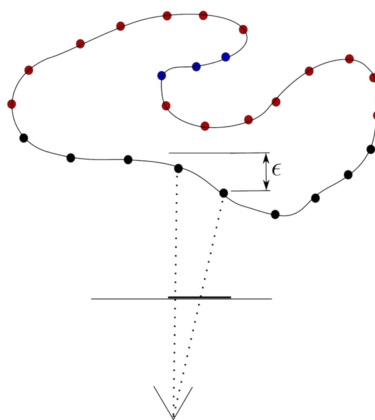


Figure 5.8: The z-buffer test must be performed with a tolerance to avoid missing neighboring splats.

in the visibility test, called an ϵ-z-buffer test (Figure 5.8). A splat is only discarded if it is farther than the z buffer value plus ϵ. This approach works well, specially because, as we work with a smooth base surface for positions, depth changes slowly.

The ϵ-z-buffer test is not directly supported by OpenGL. While this test can be easily implemented in a pixel shader, we followed a different approach presented in [67]. As described before, during a first pass we render large splats to fill the depth buffer. However, during the second pass, we move all splats an ϵ distance in the direction of the camera. Therefore, any splat in the tolerance range will pass the visibility test.

Having resampled the points in the editing chart, we proceed with a simple approach for interpolation. Each unknown image pixel receives the attribute of its nearest neighbor. In fact, we use nearest neighbor in an L_∞ norm because it is easier to calculate using a breadth-first search. From a splatting point of view, this nearest neighbor approach behaves as if we were using anisotropic splats. Additionally, more advanced interpolation [69] or anisotropic splats could be used. In the latter approach, the size and orientation of each splat could be precisely determined from the original parametrization tangent vectors.

Many operations work in small disk neighborhoods of a point. While this is well defined for manifolds, it is not for manifolds with border. As an example, consider filtering a signal on point in the border of a surface. For instance, this border could be caused by mouth, nose or eye holes in a face. We would like to perform a convolution, but there is simply no signal on the other side of the border. In fact, this is the same problem faced by the signal processing field when filtering images, which can be regarded as manifolds with border. Their common approach is extending the signal either by using the nearest value or by mirroring. In this work, we use the editing chart to extend the surface signal (Figure 5.9). Besides interpolating, the BFS filling algorithm is used for extending the signal, each pixel in a point's neighborhood receives the color of its nearest neighbor (Figure 5.10). Although, the attributes can be extended indefinitely, we cannot do the same with the ids. The reason is the ids will be used for later resampling, as such they should

(a) Signal

(b) Signal Extension

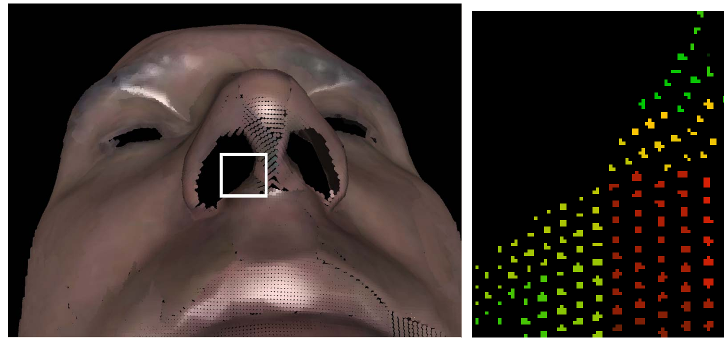
Figure 5.9: We use a BFS filling on the editing chart to extend the surface signal.

only represent the area of influence of the original texel. This limited region of influence is accomplished limiting the depth of the BFS. In fact, there is no need for very large extensions of the attributes either, they must only be extended as far as the support of the operation.

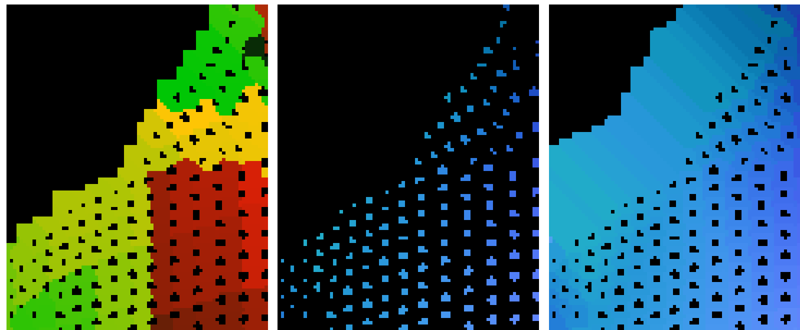
Multiresolution Projective Atlas

We chose to implement the storage atlas as a projective atlas [38]. This atlas has some advantages. First, instead of storing position images, we only store depth images in each chart. This approach saves space compared to QSplat [68] which stores position for each point or compared to geometry images [3] which stores position for each texel. The reason storing only depth images works is that it is possible to recover the actual position from the depth and the camera parameters, which must be stored separately. The process is simple: trace a ray from the camera center through the center of each texel and walk the stored depth along this ray. In other words, geometry, in this case depth, together with camera parameters let us calculate the parametrization function.

In fact, we used a *multiresolution* projective atlas [70]. Using the charts, it is simple to build a multiresolution model: each image is downsampled multiple times. This mip-mapped atlas induces a multiresolution point cloud. We use this repre-



(a) Shaded (b) Id



(c) Id Extension (d) Normal (e) Normal Extension

Figure 5.10: The nose hole is a border of the surface. For this reason, the attributes must be extended to allow operations in larger neighborhoods such as filtering.

Figure 5.11: A multiresolution atlas supports progressive rendering. While a low resolution version is exhibited, high resolution models can be loaded in the GPU, avoiding delays in the change from editing to viewing mode.

sentation to support progressive rendering in the viewing mode. With progressive rendering (Figure 5.11), the system can switch automatically to lower resolution versions of the model to meet a desired framerate. While a low resolution version is exhibited, high resolution models can be loaded in the GPU, avoiding delays in the change from editing to viewing mode.

In future work, we would like to use this multiresolution atlas for editing purposes directly [71]. For instance, when editing from a distance, texels are minified. Minified samples could have an editing semantics of only editing low-scale version of the signals. Computationally, we would use splats at different scales to induce an adaptive sampling of the editing chart. In this scenario, adaptivity would happen both in space and frequency.

5.3.3 Storage Update

To update storage charts, we need a method to resample the editing chart. As described before, we chose to use an id-buffer. We have already discussed the construction of the id-buffer, both in the context of triangle rendering and of point rendering. Next, we show how to use the id-buffer to actually perform the resampling of the editing chart back to the storage charts.

During the rendering of the editing chart two situations may occur: the samples may be magnified, a texel occupies many pixels, or minified, a pixel contains many texels. The resampling method we developed is quite simple and only works for magnified samples. However, this is not a big restriction since we have control over the editing chart construction. We achieve our best results when there is one empty pixel between samples.

We have implemented two simple approaches to resampling: point and area sampling. For point sampling we copy the attribute of the center of the projected texel back to the storage chart. In the case of area sampling, the average value of the attribute over all pixels in the texel's cell is used.

In future work, multiresolution samples can be used to handle editing in minified charts. In other words, we would work with texels at many resolutions. A screen

pixel could be related to larger region of a model represented by a low resolution texel. Additionally, for charts containing magnified samples, the simple resampling approaches presented may lead to signal loss, e.g, very high frequency content simply will not be represented in the smaller storage grid. Carr et al. [56] use an adaptive atlas allocating more texture resolution according to the signal. On the other hand, in Pointshop3D [57], the authors simply create more point samples in their point cloud. In this context, this is an easy operation because point clouds are unstructured.

5.4 Chart Distortion

In our method, all editing operations are processed in an orthogonal editing chart. However, given a chart we cannot edit the signal at each visible point. First because the orthogonal mapping introduces distortions in the texture, as can be seen comparing the relative sizes of the red and blue rectangles (Figure 5.12). The reason is the orthogonal projection is only isometric when the surface's normal is facing the camera. Even though this is seldom true, we can still work with low distortion charts by only working in regions where the normals are far from being orthogonal to the camera's viewing direction. Another example of texture distortion is shown in Figure 5.13, in addition it shows a measure of distortion by color coding the mapping derivative norm.

Second, working in regions where the normal makes a small angle with the camera direction largely improves sampling. Parametric distortions are a major concern for any resampling procedure because they lead to texture quality loss. In particular, the id buffer approach we use for resampling is vulnerable. As described before, the reason is we do not handle the case of samples being minified, in other words a pixel is occupied by at most one texel. This tends to happen when normals are far from the camera direction and thus samples get very crowded, occupying only a few pixels. As a result, our id buffer would be incomplete missing texels which would not get updated.

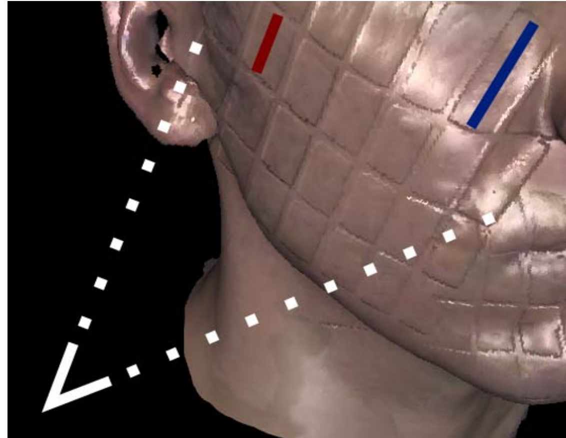


Figure 5.12: This example shows a pattern applied from the white camera point of view. A projective editing chart introduces distortions in the texture, as can be seen comparing the relative sizes of the red and blue rectangles.

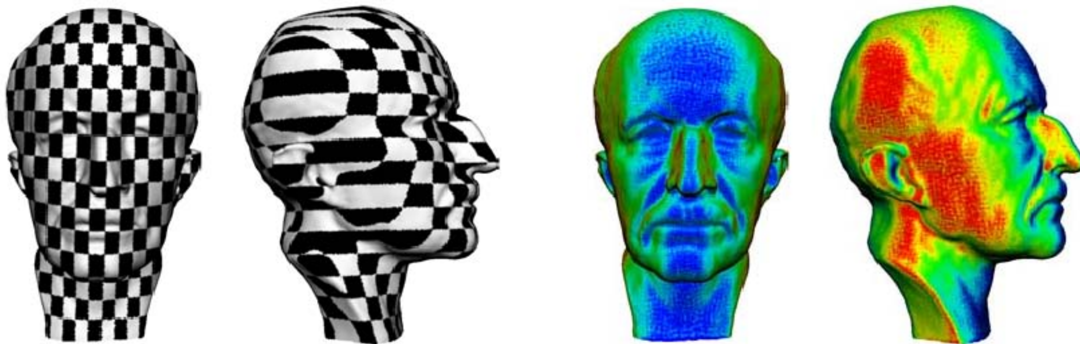


Figure 5.13: An example of texture distortion caused by orthogonal charts. We show a measure of distortion by color coding the mapping derivative norm. Image taken from [72].

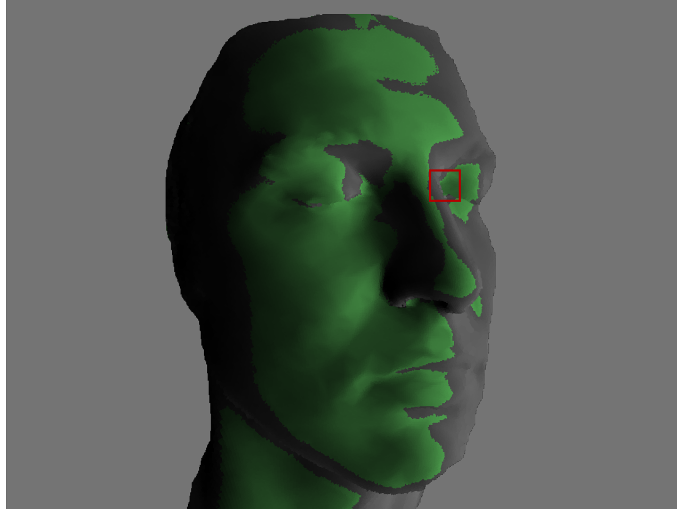


Figure 5.14: We only allow editing in a subset of the visible region, namely regions where the normal makes a small angle with the camera direction. In addition, we cannot edit near depth discontinuity, shown by the red square. A threshold on normal angle and distance from discontinuities defines the green region shown above.

The solution to both problems above is only allowing editing in a subset of the visible region, namely regions where the normal makes a small angle with the camera direction (Figure 5.14). We call this region the effective chart. In this work, the user selects the most appropriate chart for editing. Most user manipulations in our work used a light in the same position as the camera. Under these conditions, large specular highlights provide excellent feedback of normal deviation.

In addition to thresholding normals, we must also avoid depth discontinuities (Figure 5.14-red). Even when there are no sampling problems near discontinuities, there is a large topological problem. In these regions, image neighborhoods do not match surface neighborhoods. Editing with this discontinuous mapping results in wrong results or discontinuities in the final attributes.

Attribute discontinuities can also come from other sources. In Figure 5.15-left, two different editing charts were used to fill the surface with different colors. This approach creates a color discontinuity. In the same image, we show how using a blending function during each filling operation produces a seamless result. Our

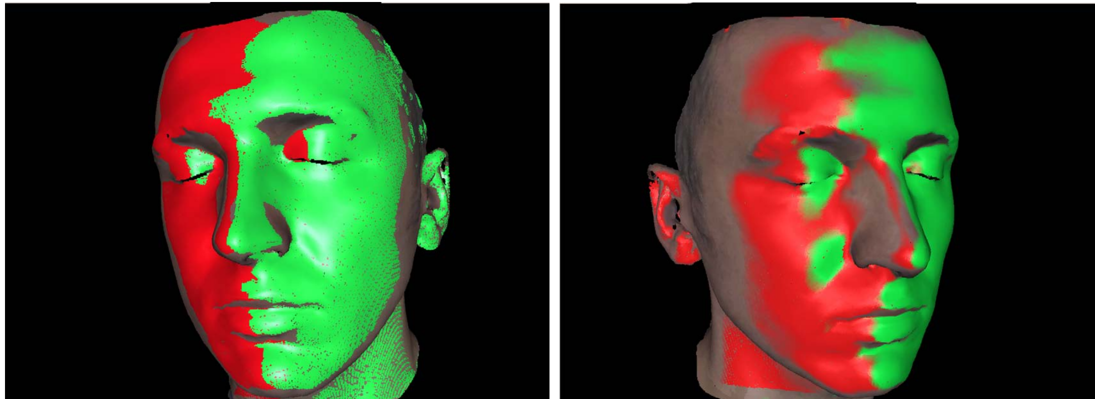


Figure 5.15: On the left, two different editing charts were used to fill the surface with different colors. This approach creates a color discontinuity. On the right, we used a blending function during each filling operation, resulting in a seamless result.

approach is simple, we build a weighting function (Figure 5.16) based on normal angles and distances to discontinuities. This function values one inside the effective chart and smoothly goes to 0 in the border. After an editing operation, we update the surface attribute using this function for blending. As can be seen in Figure 5.15-right, the result is seamless.

Thresholding normals can also be used with an editing semantics. When physically spraying paint on a surface, more paint will tend to fall on regions of the surface that face the spray can. In Figure 5.17, we show how this can be emulated by using the normal angle as blending function. The red ink was selectively applied to the model based on the existing normals. In addition, the blue result applied normals and colors at the same time. One limitation of this technique is that we cannot chose whether to apply paint based on depth, only normal angles.

5.5 Conclusion

In this chapter, we presented a computational solution for editing surface signals with editing charts. We have used a texture atlas to store attributes but an additional temporary editing chart for operations. Two computational solutions were

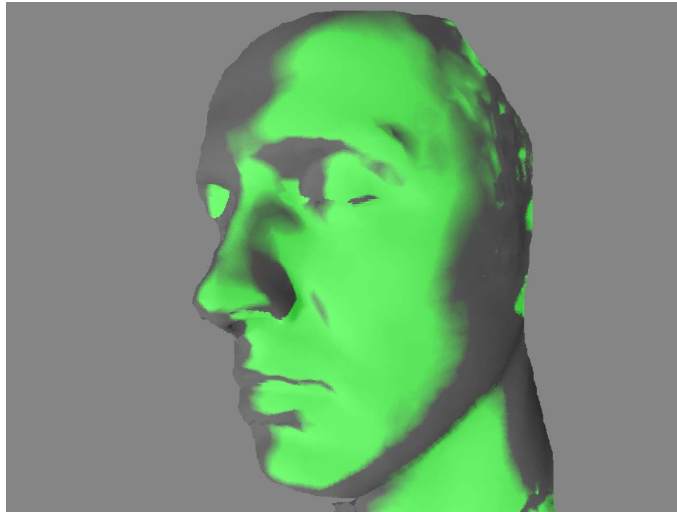


Figure 5.16: In green we show a weighting function based on normal angle and distance to discontinuity. This function is used to blend with existing attributes.

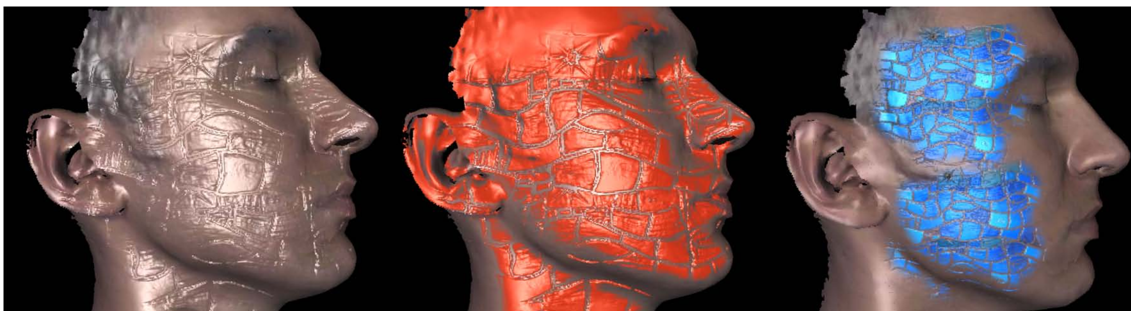


Figure 5.17: The red ink was selectively applied to the model based on the angle between the existing normals and the camera. In addition, the blue result applied normals and colors at the same time.

presented for transferring information between texture charts. While one uses triangles as primitive, the other uses points. In addition, we have seen how this transfer framework together with all modeling operations can be integrated in an interactive system.

The design presented in this chapter satisfies the computational requirements posed in the introduction for a mesostructure modeling system:

- Interactive editing;
- Workspace with realtime lighting feedback (WYSIWYG);
- Representation of very high resolution details.

Further optimizations should allow very high resolution models. For example, as remarked, it should be possible to implement this design in the GPU. Most of the method is image-based and largely parallelizable. In addition, improved sampling and reconstruction methods are necessary for the point based transfer solution. The multiresolution point structure should help in designing fast and accurate resampling solutions.

Chapter 6

Conclusion

This chapter concludes this thesis by summarizing its main contributions, by discussing the advantages and limitations of the chosen approach, and by presenting future work possibilities.

6.1 Principal Contributions

In this work, we focused on representing and processing normals in geometric surfaces. We next list our main contributions:

- We presented methods for **processing** normals in RGBN images including any linear filter, linear and non-linear combination, normal image warping and sketch-based sculpting. We presented a theoretical analysis of these operators to guarantee control and show properties of the resulting surface.
- We developed a **normal synthesis** pipeline on RGBN images. In particular, the method separates exemplar and models into base shape and details to have frequency control during synthesis.
- We presented formal **surface normal operations**. For this task, we used a base surface and detail separation to formalize a normal mapped surface. While the base surface is represented by a mesh, details are only stored as normals. In addition, we introduced the **temporary orthogonal chart** in the

context of normal editing. This mapping allows us to define normal operations on surfaces both for view-dependent and normal displacement editing. This includes all previous RGBN image operations.

- An **interactive normal editing system** that implements the data transfer operations necessary to work with orthogonal editing charts given a texture atlas. In this context, we present two transfer solutions: one with normal mapped triangles and one **point-based transfer** method. Our system uses a mesh with a texture atlas to work with high resolution details on surfaces of any topology.

6.2 Discussion

We have demonstrated that normals are a useful representation for details including modeling operations. Modeling geometric details with normals leads to a number of advantages:

- Using normals for details provides a **unified representation** for acquisition, processing and rendering of 3D objects.

- In acquisition, normals can be captured using photometric approaches and processed directly. There is no need to convert between different surface representations, e.g., a high resolution mesh.

- In real-time rendering, previous approaches model details with a mesh and next project them on a normal map. Instead, our approach works with a normal mapped model directly. As a consequence, in our method what you see is what you get (WYSIWYG).

- Some operations are more naturally specified with normals, in particular, tangent plane operations, gradient domain operations [41] and shading-based editing [73].

- The editing chart allows image-based methods to be applied to processing normals in surfaces leading to fast geometry processing algorithms. Algorithms previously proposed for images can be directly used for surfaces. In particular, we have shown how bilateral filters and texture synthesis methods can be used for manipulating details in surfaces.

- Normals are a compact representation for surface details as they can be stored in textures. This impacts not only storage but also algorithm efficiency. First, textures are easier to render because of smaller graphics bandwidth requirements. Second, textures are easier to process due to implicit connectivity. Overall, this representation lets interactive modeling systems handle large models efficiently.

- As normals are already derivatives, they should be an appropriate representation for sharp features. However our present point-based rendering method limits these applications.

On the other hand, our surface normal processing has a number of limitations that must be considered for applications:

- Working in orthogonal charts involves frequent resampling operations that can lead to a loss in attribute quality specially for textures and sharp features, both very important for details.

- Most of the mathematical models presented require a base surface of low curvature and differentiable detail scalar displacement. This imposes restrictions on the surface or requires a separation in a pre-processing step. However, there is likely a gap between theory and implementation since our system could be directly applied to non-smooth base surfaces and/or non differentiable details, such as displacements encoding sharp features.

- The details must be represented as a scalar displacement or a height field, since most of our operations convert normals to a gradient of a real function.

- Orthogonal charts cannot map larger regions than can be contained in a view. This can be a problem when editing a large region of an object. Multiple charts would have to be used, which brings additional complications.
- We have shown that many geometric operations are possible when working directly with normals. However, most geometric operations are currently defined with positions and thus are not available for processing details represented as normals.

6.3 Future Work

In our work, we presented a few normal brushes including view-dependent and base normal displacement brushes. We have also discussed how to preserve or destroy existing details. However, current mesostructure processing tools provide other kinds of brushes and/or blending modes. In future work, we would like to develop additional sculpting operations. One particular problem is that most blending formulations require knowledge of displacements, which were assumed unavailable in this work.

New normal operations or interaction modes are made possible by modern user interfaces. Even if we restrict ourselves to the problem of specifying a single normal vector, new interface technologies bring new possibilities.

In the context of geometry acquisition, building our representation from captured data still requires careful calibration and simultaneous acquisition of macro and mesostructure. In future work, it should be interesting to use photogrammetric texture mapping methods [74] to map photometric normal to pre-existing models. To make acquisition easier, an important advance would be the automatic registration [75] of mesh and RGBN images reducing the calibration burden. In addition, fusion of multiple RGBN images should enable not only more accurate surfaces but also larger and larger RGBN images. Dynamic mesostructure in the form of RGBN-t, i.e, videos with color and normal per pixel, are already available [76]. As a consequence, manipulation methods that handle time would be useful. Present

capture techniques still require expensive hardware, therefore more practical capture methods are necessary for its wide spread use.

In future work, we would like to extend our normal processing operations to parametrizations other than orthogonal projections. This would avoid all resampling during editing chart construction specially for local operations that do not profit from the editing chart support of big neighborhoods.

Some mesh processing operations are easier to specify using normals. In particular, operations based in specifying shading are more naturally formulated with normal deformations maybe obtained from shape from shading algorithms [77, 73]. However, to leverage the power of normal operations in this context, we would need a method to change position to reflect changes in normals. In other words, we need a normal integration engine. A big challenge would be getting it to work interactively similarly to the interactive gradient painting method proposed in [20].

With conversion from normals to positions, it would be possible to build a dynamic and adaptive separation of macro and mesostructure. This representation could adjust the scale separation accordingly. For example, it could use height displacements to enhance a mesh near silhouettes, but use normals in smooth parts to save resources. In addition, macrostructure operations should be integrated, not only large scale deformations but also topological changes. The natural extension of this framework would be a representation that transparently converts geometric information between micro, meso and macrostructures according to the target level of detail.

In conclusion, we have presented operations for processing surface normals. These operations enhance the classical separation of base surface and details allowing a unified representation for acquisition, modeling and real-time rendering. Future work should profit from the introduced formalism in designing more advanced operations.

Bibliography

[1] Denis Zorin, Peter Schröder, and Wim Sweldens. Interactive multiresolution mesh editing. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 259–268, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[2] 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.

[3] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. *ACM Trans. Graph.*, 21(3):355–361, 2002.

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

[5] Robert L. Cook. Shade trees. *SIGGRAPH Comput. Graph.*, 18(3):223–231, 1984.

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

[7] 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.

[8] 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.

[9] 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.

[10] Robert J. Woodham. Photometric method for determining surface orientation from multiple images. pages 513–531, 1989.

[11] Fausto Bernardini, Holly Rushmeier, Ioana M. Martin, Joshua Mittleman, and Gabriel Taubin. Building a digital model of Michelangelo's Florentine Pieta. *IEEE Computer Graphics and Applications*, 22(1):59–67, 2002.

[12] 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.

[13] 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.

[14] 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.

[15] Thiago Pereira and Luiz Velho. Editing RGBNs. Visgraf - Vision and Graphics Laboratory - IMPA, 2009.

[16] Pixologic. Online documentation. *http://www.pixologic.com.*

[17] 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.

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

[19] Mauricio Bammann Gehling, Christian Hofsetz, and Soraia Raupp Musse. Normalpaint: an interactive tool for painting normal maps. *Vis. Comput.*, 23(9):897–904, 2007.

[20] James McCann and Nancy S. Pollard. Real-time gradient-domain painting. *ACM Trans. Graph.*, 27(3):1–7, 2008.

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

[22] Michael Toksvig. Mipmapping normal maps. *journal of graphics tools*, 10(3):65–71, 2005.

[23] Charles Han, Bo Sun, Ravi Ramamoorthi, and Eitan Grinspun. Frequency domain normal map filtering. *ACM Trans. Graph.*, 26(3):28, 2007.

[24] Emilio Vital Brazil, Thiago Pereira, Ives Macedo, Luiz Velho, Luiz Henrique de Figueiredo, and Mario Costa Sousa. RGBN sketch-based image warping. submitted for publication, 2010.

[25] Manfredo Do Carmo. *Differential Geometry of Curves and Surfaces.* Prentice Hall, 1976.

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

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

[28] 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*, Washington, DC, USA, 1999. IEEE Computer Society.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Antonio Haro, Irfan A. Essa, and Brian K. Guenter. Real-time photo-realistic physically based rendering of fine scale human skin structure. *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, 2001.

[34] Jan Kautz, Wolfgang Heidrich, and Hans-Peter Seidel. Real-time bump map synthesis. *HWWS '01: EUROGRAPHICS workshop on Graphics hardware*, 2001.

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

[36] Thiago Pereira and Luiz Velho. RGBN image editing. *Proceedings of SIBGRAPI*, 2009.

[37] David M. Mount. ANN programming manual. Technical report, 1998.

[38] Luiz Velho and Jonas Sossai Jr. Projective texture atlas construction for 3d photography. *Vis. Comput.*, 23(9):621–629, 2007.

[39] Olga Sorkine, Yaron Lipman, Daniel Cohen-Or, Marc Alexa, Christian Rössl, and Hans-Peter Seidel. Laplacian surface editing. In *Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pages 179–188. ACM Press, 2004.

[40] Andrew Nealen, Olga Sorkine, Marc Alexa, and Daniel Cohen-Or. A sketch-based interface for detail-preserving mesh editing. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 24(3):1142–1147, 2005.

[41] Yizhou Yu, Kun Zhou, Dong Xu, Xiaohan Shi, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Mesh editing with poisson-based gradient field manipulation. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 644–651, New York, NY, USA, 2004. ACM.

[42] Yaron Lipman, Olga Sorkine, Daniel Cohen-Or, David Levin, Christian Rössl, and Hans-Peter Seidel. Differential coordinates for interactive mesh editing. In *Proceedings of Shape Modeling International*, pages 181–190. IEEE Computer Society Press, 2004.

[43] Olga Sorkine and Marc Alexa. As-rigid-as-possible surface modeling. In *Proceedings of Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pages 109–116, 2007.

[44] D. Zorin, P. Schröder, A. DeRose, L. Kobbelt, A. Levin, and W. Sweldens. Subdivision for modeling and animation. In *SIGGRAPH '00: ACM SIGGRAPH 2000 Courses*, New York, NY, USA, 2000. ACM.

[45] Denis Zorin. Modeling with multiresolution subdivision surfaces. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, pages 30–50, New York, NY, USA, 2006. ACM.

[46] Autodesk. Online documentation. *http://usa.autodesk.com/*.

[47] Lexing Ying, Aaron Hertzmann, Henning Biermann, and Denis Zorin. Texture and shape synthesis on surfaces. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 301–312, London, UK, 2001. Springer-Verlag.

[48] Henning Biermann, Ioana Martin, Fausto Bernardini, and Denis Zorin. Cut-and-paste editing of multiresolution surfaces. *ACM Trans. Graph.*, 21(3):312–321, 2002.

[49] Henning Biermann, Ioana M. Martin, Denis Zorin, and Fausto Bernardini. Sharp features on multiresolution subdivision surfaces. *Graph. Models*, 64(2):61–77, 2002.

[50] Kai Hormann, Bruno Lévy, and Alla Sheffer. Mesh parameterization: theory and practice. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 1, New York, NY, USA, 2007. ACM.

[51] P. V. Sander, Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. In *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 146–155, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[52] Aaron Lee, Henry Moreton, and Hugues Hoppe. Displaced subdivision surfaces. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 85–94, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[53] Rony Zatzarinni, Ayellet Tal, and Ariel Shamir. Relief analysis and extraction. In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, pages 1–9, New York, NY, USA, 2009. ACM.

[54] Venkat Krishnamurthy and Marc Levoy. Fitting smooth surfaces to dense polygon meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 313–324, New York, NY, USA, 1996. ACM.

[55] Pat Hanrahan and Paul Haeberli. Direct wysiwyg painting and texturing on 3d shapes. *SIGGRAPH Comput. Graph.*, 24(4):215–223, 1990.

[56] Nathan A. Carr and John C. Hart. Painting detail. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 845–852, New York, NY, USA, 2004. ACM.

[57] Matthias Zwicker, Mark Pauly, Oliver Knoll, and Markus Gross. Pointshop 3d: an interactive system for point-based surface editing. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 322–329, New York, NY, USA, 2002. ACM.

[58] Cindy M. Grimm and John F. Hughes. Modeling surfaces of arbitrary topology using manifolds. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 359–368, New York, NY, USA, 1995. ACM.

[59] Cindy Grimm and Denis Zorin. Surface modeling and parameterization with manifolds: Siggraph 2006 course notes. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, pages 1–81, New York, NY, USA, 2006. ACM.

[60] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. Least squares conformal maps for automatic texture atlas generation. *ACM Trans. Graph.*, 21(3):362–371, 2002.

[61] Alla Sheffer and John C. Hart. Seamster: inconspicuous low-distortion texture seam layout. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 291–298, Washington, DC, USA, 2002. IEEE Computer Society.

[62] Dan Piponi and George Borshukov. Seamless texture mapping of subdivision surfaces by model pelting and texture blending. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 471–478, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[63] Tolga Tasdizen, Ross Whitaker, Paul Burchard, and Stanley Osher. Geometric surface processing via normal maps. *ACM Trans. Graph.*, 22(4):1012–1033, 2003.

[64] Wei D Yagou H. High-boost mesh filtering for 3-d shape enhancement. In *Journal of Three Dimensional Images*, 2003.

[65] Zhi-yang Chen Yin Zhang Xiu-zi Ye Jian-guo Shen, San-yuan Zhang. Mesh sharpening via normal filtering. In *Journal of Zhejiang University*, 2009.

[66] Kai-Wah Lee and Wen-Ping Wang. Feature-preserving mesh denoising via bilateral normal filtering. In *CAD-CG '05: Proceedings of the Ninth International Conference on Computer Aided Design and Computer Graphics*, pages 275–280, Washington, DC, USA, 2005. IEEE Computer Society.

[67] Markus Gross and Hanspeter Pfister. *Point-Based Graphics (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[68] Szymon Rusinkiewicz and Marc Levoy. Qsplat: a multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 343–352, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[69] Ricardo Marroquim, Martin Kraus, and Paulo Roma Cavalcanti. Efficient point-based rendering using image reconstruction. In *Symposium on Point-Based Graphics 2007, Prague-Czech Republic*, September 2007.

[70] Carlos Caballero and Luiz Velho. Multi-resolution PBR data structure for projective atlases. In *Universidad Michoacana*, 2009.

[71] Ken Perlin and Luiz Velho. Live paint: painting with procedural multiscale textures. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 153–160, New York, NY, USA, 1995. ACM.

[72] Mark Pauly. Point primitives for interactive modeling and processing of 3d geometry. *PhD Thesis*, 2003.

[73] Holly Rushmeier, Jose Gomes, Laurent Balmelli, Fausto Bernardini, and Gabriel Taubin. Image-based object editing. In *In Proc. 3DIM '03*, pages 20–28. Society Press, 2003.

[74] Yochay Tzur and Ayellet Tal. Flexistickers: photogrammetric texture mapping using casual images. *ACM Trans. Graph.*, 28(3):1–10, 2009.

[75] Thales Vieira, Adelailson Peixoto, Luiz Velho, and Thomas Lewiner. An iterative framework for registration with reconstruction. In *Vision, Modeling, and Visualization 2007*, pages 101–108, Saarbrücken, november 2007. Pirrot.

[76] Pieter Peers-Charles-Felix Chabert Malte Weiss-Paul Debevec Wan-Chun Ma, Tim Hawkins. Rapid acquisition of specular and diffuse normal maps from polarized spherical gradient illumination. *Eurographics Symposium on Rendering*, 2007.

[77] Yotam Gingold and Denis Zorin. Shading-based surface editing. *ACM Trans. Graph.*, 27(3):1–9, 2008.