Dissertação para obtenção do grau de Mestre em Matemática pelo
INSTITUTO NACIONAL DE MATEMÁTICA PURA E APLICADA

Variational Texture Atlas Construction and Applications

por
JONAS SOSSAI JÚNIOR

Orientador: LUIZ VELHO

29 de Maio de 2006

Abstract

The use of attribute maps for 3D surfaces is an important issue in Geometric Modeling, Visualization and Simulation. Attribute maps describe various properties of a surface that are necessary in applications. In the case of visual properties, such as color, they are also called texture maps.

Usually, the attribute representation exploits a parametrization $g\colon U \subset \mathbb{R}^2 \to \mathbb{R}^3$ of a surface in order to establish a two-dimensional domain where attributes are defined. However, it is not possible, in general, to find a global parametrization without introducing distortions into the mapping. For this reason, an atlas structure is often employed. The atlas is a set of charts defined by a piecewise parametrization of a surface, which allows local mappings with small distortion.

Texture atlas generation can be naturally posed as an optimization problem where the goal is to minimize both the number of charts and the distortion of each mapping. Additionally, specific applications can impose other restrictions, such as the type of mapping. An example is 3D photography, where the texture comes from images of the object captured by a camera [7]. Consequently, the underlying parametrization is a projective mapping.

In this work, we investigate the problem of building and manipulating texture atlases for 3D photography applications. We adopt a variational approach to construct an atlas structure with the desired properties. For this purpose, we have extended the method of Cohen-Steiner *et al.* [9] to handle the texture mapping set-up by minimizing distortion error when creating local charts. We also introduce a new metric tailored to projective maps that is suited to 3D photography.

Projective texture atlas serves as a foundation for an attribute processing framework. We exploit it in the user interface of a texture editing/painting interactive application. Other features incorporated into this framework include: texture compression, blending and inpainting. Our current research is looking into using surface attributes like normal and displacement fields for modeling operations.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation and Problem Description

In the quest of reconstruction of high-quality 3D models we have to give attention to both surface shape and reflectance attributes. Most high-end 3D scanners sample the surface shape in a very high resolution. Therefore the difference between a good and a bad reconstruction is how the surface reflectance attributes, such as color, normal, specularity, illumination, etc, are captured and applied to the model.

Texture mapping, a special case of attribute mapping in which the attributes are visual properties, is a powerful tool to represent these surface attributes. This technique is based on mapping an image (either synthesized or digitized) onto a given surface. A mapping is a function that assigns a pair of coordinates referring to a pixel of the planar image to each point of a surface.

Mapping a 2D texture onto a 3D surface requires a parametrization of the surface. This parametrization is obtained easily when the surface is defined parametrically, which is not the case of the most 3D meshes in Graphics. And even if we had got this global parametrization, we would have many problems related to distortions, singularities, etc.

A solution to these problems of to use a atlas structure, a set of charts defined by a piece-wise parametrization. Depending on how these charts are constructed we can obtain local mappings with small distortion. This approach was explored in several areas in Graphics: 3D photography [7], painting [8], surface representation [23], etc.

The problem of constructing a texture atlas could be set as a optimization problem, where the goal is to minimize the number of charts of the atlas, as well as the total distortion of the mapping. There are many works that deal with the problem of finding a good surface partition. In Sander *et al.* [23] a greedy face clustering algorithm was proposed, which does not guarantee an optimal partition. Although Cohen-Steiner *et al.* [9] base their partition through clustering too, they seek a partition that minimizes a given error metric (a variational partition).

Given this idea of constructing a texture atlas using a variational surface partitioning method, we explore two important areas of computer graphics:

- **3D photography**: Following the pipeline described by Callieri *et al.* [7] we develop a method for generating multiresolution texture maps from a set of images. With the

variational scheme we are able construct a well partitioned and low distorted texture atlas.

- **Attribute Editing**: The variational scheme help us to created texture atlases to be used by a intelligent and powerful painting system, which allows the user to manipulate such as texture atlases.

1.2 Related Work

In our work we have explored many areas in Geometric Modeling and Visualization, like 3D photography, atlas construction and painting. There has been a significant amount of work done in these areas.

In 3D photography, some important works were developed for texture reconstruction from a set of images. One the first works that cover all the process of reconstructing both geometry and texture from multiple scans was the work of Bernardini *et al.* [3]. Other important work, focused in texture map construction, was the work of Callieri *et al.* [7], which partitions the surface in regions (in a greedy fashion) and constructs an image-to-surface mapping that minimizes the final texture map distortion.

The problem of atlas construction has been proposed for different areas. In texture map construction, the works of Callieri *et al.* [7] (as we saw in the previous paragraph) and Maillot *et al.* [19], which partitions a mesh into charts based on bucketing of face normals, are distinguished. Sander *et al.* [23] have used this atlas scheme for surface representation. Carr and Hart [8], in other way, proposed a multiresolution texture atlas for painting. In Lévy *et al.* [17] texture atlases for 3D painting systems are constructed.

One of the most important works in surface painting and texture mapping was developed by Hanrahan and Haeberli [13]. In this work they have developed a powerful interface for painting colors, materials and lighting properties. Other important work in this area was proposed by Carr and Hart [8], in which a multiresolution texture atlas was used for painting.

1.3 Contributions

We propose a method for generating projective texture atlases using the variational surface partition scheme developed by Cohen-Steiner *et al.* [9] (different from other approaches that construct texture atlases in a greedy fashion) and explore applications for 3D photography and painting. Since our atlases are projective, the underlying parametrization comes from the projective transformation of a camera associated with each chart.

In 3D photography, based on the pipeline described by Callieri *et al.* [7], we develop a method for texture map construction using the variational scheme to generate a projective texture atlas. In particular, we explore two aspects in the process: construct local charts with small distortion and similar texture detail content. In addition, we develop a new diffusion method to eliminate the color discontinuity between adjacent charts due to non-uniform illumination conditions of the photographs.

In painting we create a attribute processing application that uses projective texture atlas as foundation. For this application we develop a variational method for constructing a texture

atlas and a set of virtual cameras based on the ideas of Cohen-Steiner *et al.* [9], Sander *et al.* [23] and Marinov and Kobbelt [21].

1.4 Structure

We organize the text as follows:

- **Chapter 2:** Introduces the variational scheme developed by Cohen-Steiner *et al.* [9] and explains how texture atlas construction can be posed as an optimization problem.
- **Chapter 3:** Explains the pipeline for texture atlas construction from multiple views and the improvements (distortion optimization, texture compression and blending).
- **Chapter 4:** Gives the structure of the attribute editing application and details the method for generating virtual cameras in a way to minimize the texture atlas distortion.
- **Chapter 5:** Shows the main results obtained by the methods developed in this work.
- **Chapter 6:** Concludes the work and lists possible future research and applications.

Chapter 2

Variational Partitioning and Texture Atlas Construction

An important step in the texture atlas construction process is how the input surface is partitioned based on a given metric or criterion. Most of methods use greedy algorithms [7, 8, 23], and an optimal partition may not be found. A solution to this problem is to use a variational scheme for partition optimization.

In section 2.1 we will describe the variational method for shape approximation developed by Cohen-Steiner *et al.* [9]. Section 2.2 shows how to use this scheme to construct texture atlases and details the process of constructing them.

2.1 The Variational Scheme for Optimization

The variational calculus is a field of mathematics which deals with functions of functions, as opposed to ordinary calculus which deals with functions of numbers. Such functionals can, for example, be formed as integrals involving an unknown function. Our interest is in extremal functions, those making the functional attain a maximum or minimum value, as the following examples:

- Find the shortest path between two points in a plane. The variables in this problem are the curves connecting the two points. The optimal solution is of course the line segment joining the points.
- Given two cities in a country with lots of hills and valleys, find the shortest road going from one city to the other. This problem is a generalization of the above, and finding this shortest road is not as obvious.
- Find a partition of a surface that is optimal in a certain sense, while satisfying some requirements.

These examples are classical problems of optimization. The last one was explored in several research themes in Graphics, most of them establishing approximation as a optimization problem. Basically an optimal partition, which gives a good approximating surface Y of a

surface X, is the one that minimizes a distance (error) function between the two surfaces. The most used metric in Graphics to measure distance, the $\mathcal{L}^p$ distance, is defined as follows:

$$\mathcal{L}^p(X,Y) = \left(\frac{1}{|X|}\iint_{x\in X} \|d(x,Y)\|^p dx\right)^{\frac{1}{p}} \tag{2.1}$$

with:

$$d(x,Y) = inf_{y\in Y}\,\|x-y\|$$

where $\|.\|$ is the Euclidean distance, while $|.|$ is the surface area. The difficulty to find this optimal partition for surface approximation (it is a NP-hard optimization problem [1]) explains the frequent use of greedy algorithms, but the restrictions that must be imposed to them (reduce the number of geometric elements at the expense of an uncontrollable approximation error or guarantee an approximation error at the expense of an uncontrollable number of geometric elements) make difficult to find good solutions in an efficient way.

To solve that, Cohen-Steiner *et al.* [9] proposed a discrete and variational partitioning method. In the next sections we will give the theoretical concepts and explain the algorithm.

2.1.1 Partition and Proxies

The problem of finding an approximation of a complex surface can be cast as a geometric partitioning one. The idea is basically to cluster geometric elements, creating in this way a partition of the surface:

A partition R of a surface $S \subseteq \mathbb{R}^3$ is a collection of k regions R_i, such that $R_i \cap R_j = \oslash$, for $i \neq j$, and $\bigcup_{i=1}^{k} R_i = S$.

Each region R_i of a partition R can be represented by a proxy:

A proxy of a region R_i is a pair $P_i = (X_i, N_i)$, with X_i the "average" point of the partition and N_i the "average" normal of the partition.

In other words, a proxy is an linear approximant for the faces of the associated region.

An example of a mesh partition and the proxies associated to the partition can be seen in Figure 2.1.

As we can see, there many ways for partitioning a surface, but we are interested in a partition that is optimal in a certain sense. In other words, we are interested in a partition that minimizes a given error/distance metric. We will define such metrics in the next section.

2.1.2 Metrics

The key to find a partition which gives an optimal approximation of the surface is defining a good error metric, based on the Equation 2.1. In Cohen-Steiner *et al.* [9] two metrics were explored: the $\mathcal{L}^2$ metric and the $\mathcal{L}^{2,1}$ metric.

Figure 2.1: A partition (left) and its associated proxies (right) (from Cohen-Steiner *et al.* [9]).

The $\mathcal{L}^2$ Metric

The $\mathcal{L}^2$ or Hausdorff metric is constantly used in Graphics when comparing two surfaces and can be extended to the definitions explained in the previous section: given a region R_i, its associated proxy $P_i = (X_i, N_i)$ and denoting $\Pi_i(\cdot)$ the orthogonal projection of the argument on the "proxy" plane going through X_i and normal to N_i, the $\mathcal{L}^2$ metric is then:

$$\mathcal{L}^2(R_i, P_i) = \iint_{x \in R_i} \|x - \Pi_i(x)\|^2 dx \tag{2.2}$$

This metric measures the squared error between the region R_i and its associated proxy P_i, and tries to match the geometry of the surface and its approximant surface. Although it is very intuitive, it fails to give a unique optimal shape in some cases (hyperbolic surfaces) and does not take into consideration the normal field. So a new metric that would circumvent these problems is essential.

The $\mathcal{L}^{2,1}$ Metric

Since normals govern lighting effects, curvature lines and silhouettes, important aspects in Visualization, a normal-based metric would be more appropriate than a position-based metric. The $\mathcal{L}^{2,1}$ was a solution proposed by Cohen-Steiner *et al.* [9] to circumvent the problems of the $\mathcal{L}^2$ metric. Given a region R_i, its associated proxy $P_i = (X_i, N_i)$, the metric can be written as:

$$\mathcal{L}^{2,1}(R_i, P_i) = \iint_{x \in R_i} \|n(x) - N_i\|^2 dx \tag{2.3}$$

Besides improving the visual results of the partition through the use of the normal field, the $\mathcal{L}^{2,1}$ metric is much simpler and, consequently, much more easier to compute. Another

important advantage is that convergence is guaranteed for convex objects for this norm [9]. The effects that $\mathcal{L}^2$ and $\mathcal{L}^{2,1}$ can have on an approximation can be seen in Figure 2.2.

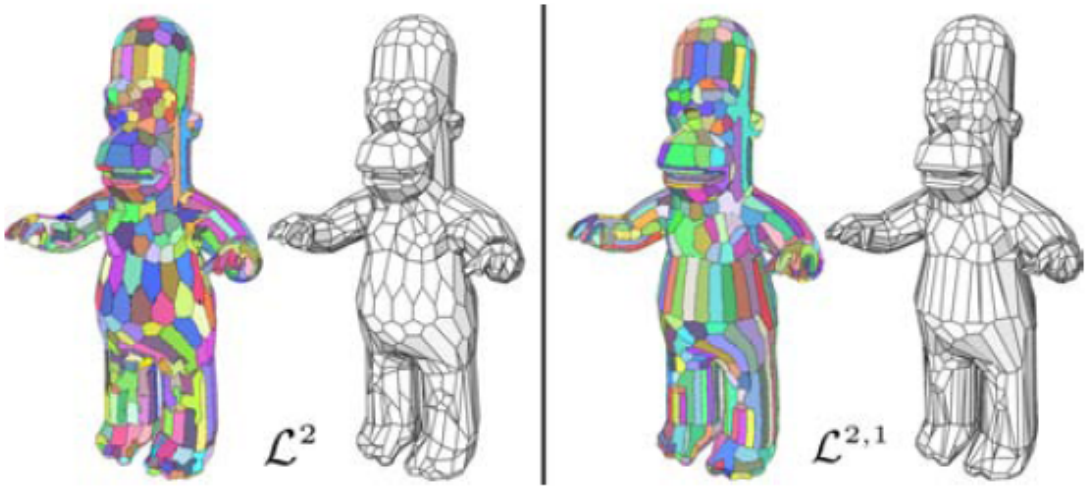


Figure 2.2: While $\mathcal{L}^2$ (left) and $\mathcal{L}^{2,1}$ (right) give similar results on near-spherical regions, the results of $\mathcal{L}^{2,1}$ on planar regions are much better (from Cohen-Steiner *et al.* [9]).

2.1.3 Optimizing Shape Proxies

Given the theoretical concepts of the previous sections, an optimal shape approximation can be defined as follows:

Given an error metric E (either $\mathcal{L}^2$ or $\mathcal{L}^{2,1}$), a desired number k of proxies, and an input surface S, we call optimal shape proxies a set P of proxies P_i associated to the regions R_i of a partition R of S that minimizes the total distortion:

$$E(R, P) = \sum_{i=1..k} E(R_i, P_i) \tag{2.4}$$

As cited in Section 2.1, this is a NP-hard optimization problem, and cannot be solved in a reasonable time. However in Cohen-Steiner *et al.* [9] the problem was set up as a discrete and variational shape approximation one, such that discrete clustering algorithms can be used to achieve very good results. A particular algorithm that was explored in Cohen-Steiner *et al.* [9] was the Lloyd's method Lloyd [18], commonly used to find the centroidal Voronoi diagram Du [10], because of the similarity of the two problems.

Centroidal Voronoi Diagram

Before giving the concepts and definitions related to the centroidal Voronoi diagram, we have to explain what a Voronoi diagram is.

Consider an open set $\Omega \subseteq \mathbb{R}^N$, with $|.|$ the Euclidean norm on $\mathbb{R}^N$. Given a set of points $\{z_i\}_{i=1}^k$ belonging to Ω, the Voronoi region V_i corresponding to the point z_i is defined by:

$$V_i = \{x \in \Omega : |x - z_i| \leq |x - z_j| \text{ for } j = 1, ..., k\} \tag{2.5}$$

The points $\{z_i\}_{i=1}^k$ are called *generators*. The set $\{V_i\}_{i=1}^k$ is called the *Voronoi diagram* of Ω and each V_i is referred to as the *Voronoi region* corresponding to z_i.

The Centroidal Voronoi Diagram is a case of the Voronoi Diagram in which the points z_i, that serve as generators for the Voronoi regions V_i, coincide with the mass centroids of those regions. Figure 2.3 shows an example of a Voronoi Diagram and a Centroidal Voronoi Diagram.

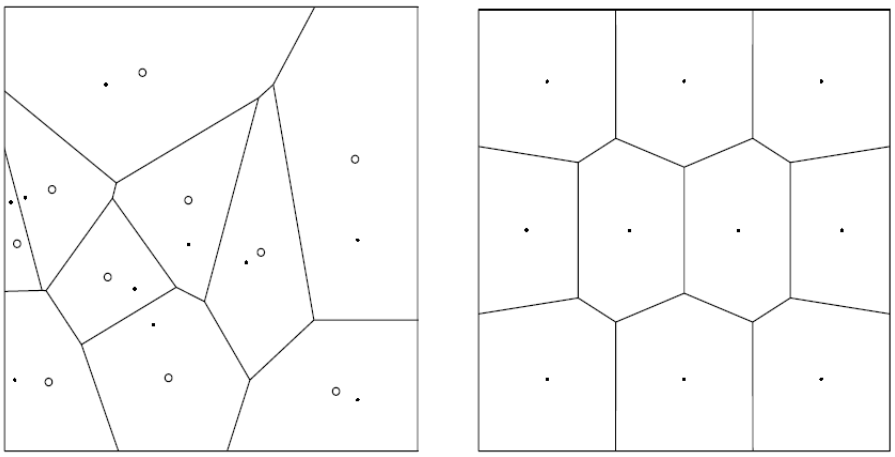


Figure 2.3: On the left, the Voronoi Diagram corresponding to 10 randomly selected points in a square. The dots are the Voronoi generators and the circles are the centroids of the corresponding Voronoi regions. On the right, a 10-point Centroidal Voronoi Diagram. The dots are simultaneously the generators for the Voronoi Diagram and the centroids of the Voronoi regions.

One may ask, how does to find a Centroidal Voronoi Diagram? A classical method to solve this problem is the Lloyd's method [18]. This method is described in Algorithm 1.

Algorithm 1 Lloyd's Method

```
Ω ← set of ℝ^N
k ← number of Voronoi regions
V_i ← Voronoi region i
z_i ← Voronoi region generator i
select an initial set of k generators {z_i}_{i=1}^k
while for some i, z_i ≠ V_i centroid do
  for i = 1 to k do
    V_i ← Voronoi region associated to z_i
    z_i ← V_i centroid
  end for
end while
```

This method aims at minimizing a cost function E based on how tightly each region is packed. The functional E, defined by a set of n points X_j and k centers c_i, is:

$$E = \sum_{i \in 1...k} \sum_{X_j \in R_i} \|X_j - c_i\|^2 \tag{2.6}$$

Variational Partitioning Algorithm

As we described in previous section, Lloyd's method can be used to minimize a cost function. Therefore we can use this method to minimize the distortion Equation 2.4 and find, in this way, a optimal partition of a surface. Cohen-Steiner *et al.* [9] developed a extension of this algorithm to variational partition that include two steps:

1. **Geometry Partitioning:** Through the use of an error-minimizing region growing algorithm, construct a partition of the surface (similar to the first step of Algorithm 1).

2. **Proxy Fitting:** For each region of the partition compute the proxy associated, that is an extension of the centroid in the original Lloyd's algorithm (similar to the second step of Algorithm 1).

As in Lloyd's method, these two steps are repeated alternately to drive the total energy down.

2.2 Texture Atlas Construction as an Optimization Problem

In the previous section we saw that finding a partition of a surface in order to approximate it can be treated as an optimization problem. The same idea can be used to generate a good texture atlas of the associated surface, where the goal is to minimize both the number of charts and the distortion of each mapping (and consequently reduce texture stretch). Since these problems are examples of optimization, we adopt a variational approach [4] when creating the charts, in order to obtain a partition that minimizes the aspects cited above, differently from other approaches [7, 8, 17, 23], which constructs texture atlases in a greedy fashion.

Mapping is described by Lévy and Mallet [16] as follows: given an open surface $\mathcal{S}$ of $\mathbb{R}^3$, a *mapping* Φ is a bijective transform that maps the surface $\mathcal{S}$ to a subset $\mathcal{D}$ of $\mathbb{R}^2$ (Figure 2.4):

$$(x, y, z) \in S \rightarrow \Phi(x, y, z) = \begin{bmatrix} \Phi^u(x, y, z) \\ \Phi^v(x, y, z) \end{bmatrix}$$

- $\mathcal{D}$ is called the *parametric* (u, v) *domain.*

- As Φ is, by definition, a one-to-one function, it has an inverse function $x = \Phi^{-1}$, called a *parametrization* of the surface:

$$(u, v) \in D \rightarrow x(u, v) = \Phi^{-1}(u, v) = \begin{bmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{bmatrix}$$

Mapping a 2D texture into a 3D surface requires a parametrization of the surface. This parametrization is obtained easily when the surface is defined parametrically, which is not the case of most 3D meshes in Graphics. And even if we got this global parametrization, is very hard to find one without introducing distortions into the mapping.

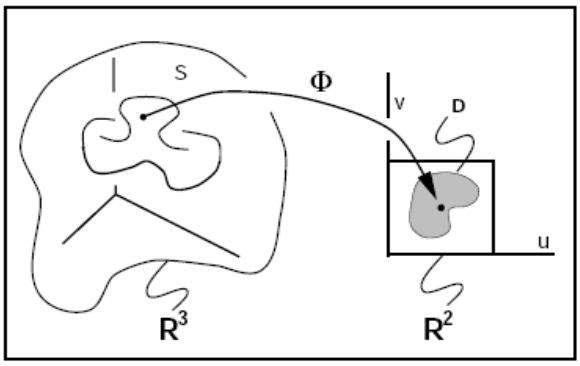


Figure 2.4: Mapping Φ from a surface $\mathcal{S}$ of $\mathbb{R}^3$ to $\mathcal{D} \subset \mathbb{R}^2$ (from Lévy and Mallet [16]).

A solution to this problem is partitioning the surface into regions in order to construct a texture *atlas*, a set of charts defined by a piecewise parametrization of the surface. In our case, a *atlas* is composed of a set of charts $\{\Theta_1, ..., \Theta_n\}$, where each chart Θ_i is an application from a patch P_i (section of the 3D mesh) to the Euclidean plane. An example of a surface partition and its associated atlas is showed in Figure 2.5.

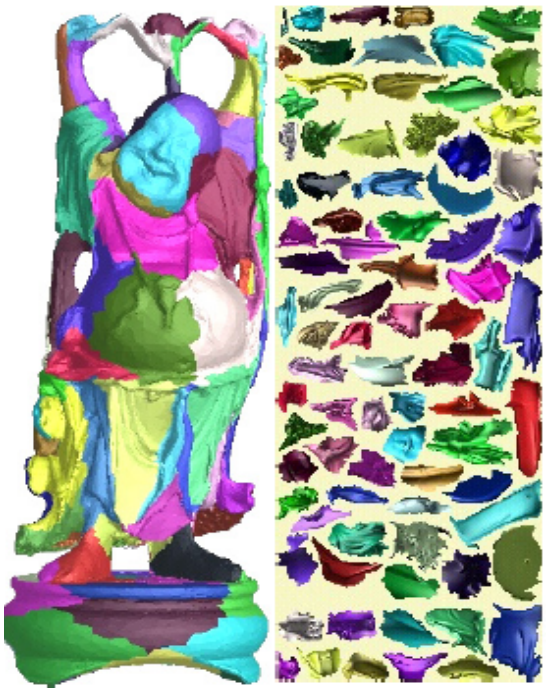


Figure 2.5: Buda model partition (left) and the associated atlas (right) (from Sander *et al.* [23]).

Basically the process of generating a texture atlas can be separated in three steps: Chart Formation, Chart Parametrization, and Chart Packing.

2.2.1 Chart Formation

The first step in the texture atlas generation process is to partition the mesh into a set of regions (patches) based on defined constraints: minimize the mapping distortion of each chart, ensure that all charts are topological disks, produce charts with compact boundaries, etc. With respect to minimize the mapping distortion of each chart, the difficulty to find this optimal partition explains the frequent use of greedy algorithms.

Sander *et al.* [23] apply a greedy face clustering algorithm based on the Lloyd's method. In their algorithm, the cost between a face F on a chart C and its adjacent face F' that is a candidate to join C is a measure of geometric distance between the two faces, and difference in normal between F' and the patch normal N_c:

$$cost(F, F') = (1 - (N_c - N_{F'}))(|P_{F'} - P_F|) \tag{2.7}$$

where N_c is the normal of the patch containing F (the average normal of all faces already in the cluster), and $P_{F'}$ and P_F are the centroid points of F' and F.

We can see in Equation 2.7 that the cost function leads in account both planarity (N_c and $N_{F'}$) and compactness ($P_{F'}$ and P_F) properties, in order to produce patches that can be treated by existing parameterization methods.

There are two problems in this work:

1. The clustering is done in a greedy fashion. In consequence an optimal partition may not be found.

2. The patches are created according not only to planarity criteria, but also compactness. For this reason, a large number of charts is generated, which introduces discontinuities between adjacent patches when constructing a texture atlas.

To overcome the first restriction, we decided to use in our chart formation algorithm the method proposed by Cohen-Steiner *et al.* [9], posing surface partition as an optimization problem. The second problem is solved with the use of a normal-based metric in the chart growing process, which tends to produce regions based on planarity criterion only.

2.2.2 Chart Parametrization

One problem that comes when we create a texture atlas is how to define a parametrization of the patches in order to minimize the texture distortion and stretch. Several schemes have been proposed to flatten surface regions to establish a parametrization. These schemes typically obtain the parametrization by minimizing an objective functional.

In our work, the parametrization when constructing a texture atlas for 3D photography and for an attribute editing application comes for free, since:

- In 3D photography (Chapter 3) the texture comes from images of the object captured by a camera. Consequently, the underlying parametrization is the inverse of the projective mapping from the camera positions.

- In attribute editing (Chapter 4) we develop a variational approach to construct a collection of virtual cameras, what means that, again, the underlying parametrization is the is the inverse of the projective mapping from the virtual camera positions.

2.2.3 Chart Packing

Once the model is decomposed into a set of parameterized charts, we can go further to the last step of texture atlas generation: chart packing. This problem is basically packing the charts into a rectangular texture image in a way to reduce the occupied space by the texture. In other words, given a set of charts, how to to find a non-overlapping placement of the charts in a way that the enclosing rectangle is of minimum area?

Although this is a NP-hard problem [22], good heuristics have been proposed in computer graphics.

Sander *et al.* [24] simplify the problem by approximating each chart with the least-area rectangle that encloses it. Since they consider compactness criterion in the charts construction, the charts are reasonably shaped, so the rectangle approximation is not too costly.

In Lévy *et al.* [17] the charts created have border with arbitrary shape (they not consider compactness criterion in the charts construction). For this reason, they develop a packing algorithm that packs the charts directly rather than their bounding rectangles, inspired by how a 'Tetris' player would operate.

We decide to use a heuristic similar to Sander *et al.* [24], but simpler, although we do not consider, like Lévy *et al.* [17], compactness criterion in our texture atlas construction algorithm. We adopt a simplified packing algorithm because our atlas construction algorithm and multiresolution analysis produces relatively small texture maps. We will detail our packing algorithm in Section 3.5.

Chapter 3

Multiresolution Texture Maps from Multiple Views

The main problem when constructing a texture mapping on a 3D model from a sequence of photos (a classical problem of 3D photography) is how to integrate the information available in the set of input images.

One of the approaches used to solve this problem is to define a partition of the surface in order to optimize the texture distortion inherent in any image-to-surface mapping. In other words, a good solution it to use an atlas structure and pose texture atlas generation as an optimization problem.

In this chapter we present our variational texture atlas approach for texture reconstruction. Assuming that we have a 3D model (more specifically, a triangle mesh) of an object and a number of photographs of it (with known camera parameters), we want to reconstruct a texture for this model with the following requirements:

- reduced texture distortion in the image-to-surface mapping,
- space-optimized texture map based on the frequency analysis of the photographs,
- reduced color discontinuity between image sections that maps on adjacent regions of the surface.

To achieve these requirements we develop a variational method to construct a texture atlas, such that each chart of the atlas is associated to a region of an input image through a parametrization that is a projective mapping, and a diffusion algorithm that uses the color and illumination difference between images assigned to adjacent charts on the surface to create smooth transitions between them. The variational method aims to optimize the surface partitioning problem with respect to a distortion-based and a frequency-based metric.

Our method follows the main steps of the process of generating texture atlases, with some modifications:

- **Surface Partitioning:** In this step we partition our input surface into a set of patches. As we said before, we pose the surface partitioning process as an optimization problem, in the way that our partitioning algorithm looks for a partition that minimizes the

number of charts, the atlas mapping distortion (through a distortion-based metric that takes into consideration the projective mapping of each camera) and the texture map occupied space (based on a frequency-based metric which uses the frequency content of the image regions associated to each chart).

- **Parameterization, Discretization and Packing:** Since we are constructing a texture atlas from a set of images, the underlying parametrization of each chart is the projective mapping of the camera associated to that chart. Once the model is decomposed into a set of parameterized charts, we can go further to the step of packing them. We simplify this problem by approximating each chart with the least-area rectangle that encloses it.

- **Texture Color Smoothing:** Due to different illumination conditions when capturing the images of the object, adjacent patches that are mapped to regions from different input images could present intense color discontinuity. To solve this problem we propose a new blending method based on the diffusion equation and multigrid computing, which diffuses the color difference between the frontier zone between adjacent patches on the whole texture.

In Section 3.1 we show the structures used in the process. Section 3.2 explains how the visibility of the mesh is determined. In Section 3.3 we describe our variational approach to construct an atlas structure for this problem. We detail in Section 3.4 our method for texture compression based on its frequency content. In Section 3.5 we show how to arrange the charts of the atlas in order to minimize the texture map occupied space. Finally, in Section 3.6 we describe our diffusion algorithm to improve color the continuity/coherence between adjacent patches.

3.1 Definitions and Objects

Before giving the details of the process of generating a texture map from multiple views, we have to explain the main structures used in the process, that also will be used by the methods described in Chapter 4.

3.1.1 The Half-Edge Representation

A common way to represent a polygon mesh is a shared list of vertices and a list of faces storing pointers for its vertices. This representation is both convenient and efficient for many purposes, however in some domains it proves ineffective.

In our case, since we are using face clustering algorithms to construct our texture atlas, we have to discover adjacency relationships between components of the mesh, such as the faces and the vertices.

One of the most common of these types of representations is the half-edge data structure [4]. The structure has this name because instead of storing the edges of the mesh it stores half-edges. Basically a half-edge is a half of a edge, is directed and the two half-edges that make up a edge have opposite directions.

To achieve the adjacency relationships that we are interested, the edges of a half-edge structure are augmented with pointers to the two vertices they touch, the two faces bordering them, and pointers to the edges which emanate from the end points.

Due to its simplicity and minimal storage, we decided to use the half-edge based library *A48*, a dynamic adaptive mesh library, developed by Luiz Velho. The library reference can be accessed in [25].

3.1.2 The View Structure

The view list is a list of the input images with associated camera definitions. Each node of the structure has the following items.

- **Image:** Input image.
- **Camera:** Camera (position, orientation, etc).
- **Laplacian pyramid:** Laplacian pyramid of the input image (Section 3.4.1).
- **Gaussian pyramid:** Gaussian pyramid of the input image (Section 3.4.1).

3.1.3 The Face Structure

A face of the mesh has a number of vectors, pointers and parameters. We give below a description of each one of these items.

- **Face:** Pointer to the face of the mesh (Section 3.1.1).
- **Area:** Face area.
- **Barycenter:** Face barycenter.
- **Normal:** Face normal.
- **Visible cameras list:** List of the cameras from which the face can be seen (Section 3.2). Each node of this list has the following parameters:
  - *View:* A pointer to the view (camera and image information) from which the face is visible.
  - *Texture coordinates:* The coordinates of the projected vertices of the face through the view camera.
  - *Angle:* the angle of acquisition (computed from the face normal and the view direction).
- **Best view index:** Index of the most orthogonal camera from the list of visible cameras (Section 3.2).
- **Patch object:** Pointer to the patch that contains the face (see Section 3.3.1).

- **Subpatch object:** Pointer to the subpatch that contains the face (see Section 3.4.3).
- **Is seed flag:** Flag indicating if the face is a seed of a patch growing iteration (Section 3.3).
- **Is patch frontier flag:** Flag indicating if the face is on the frontier of its patch (Section 3.3).
- **Level:** Level in the Laplacian pyramid (Section 3.4.2).

3.1.4 The Patch Structure

A patch is basically a section of the 3D mesh. The items of this structure are detailed below.

- **Patch:** Pointer to the patch of the mesh (Section 3.1.1).
- **Area:** Patch area.
- **Barycenter:** Patch barycenter (average of the triangles' barycenters).
- **Normal:** Patch normal (area-weighted average of the triangles' normals).
- **View:** Pointer to the associated view.
- **Bounding Box 2D (Input Image):** Bounding box of the patch, when projected onto an input image using the parameters of its associated camera.
- **Bounding Box 2D (Texture Map):** Bounding box of the patch in the texture map.
- **Bounding Box 3D:** Bounding box of the patch, when transformed using the orthogonal transformation from its associated virtual camera (Section 4.2.2).
- **Face list:** List of the faces of the patch.
- **Distortion:** Distortion or error of the patch.
- **Level:** Level in the Laplacian pyramid (Section 3.4.2).
- **Subpatches:** List of subpatches (Section 3.4.3).

3.2 Visibility Calculation

The first problem that we have to take into consideration in the construction of a texture map from a set of images is the visibility problem. Of course we should only map a particular image onto the portions of the object that are visible from its original camera viewpoint. Unfortunately the OpenGL implementation of projective texture mapping does not perform such visibility tests (the texture is mapped onto occluded polygons). Therefore visibility is a problem that we have to solve on our own.

There are several algorithms that solve the problem of visibility calculation (z-buffer, ray tracing, list priority algorithms, area subdivision algorithms, etc). For simplicity and efficiency purposes, we decide to implement an extension of the z-buffer algorithm, based on the hardware-accelerated OpenGL rendering.

For each face in the model we determine from which cameras it is visible as follows:

1. Assign each face an index.

2. For each view position, render the original faces of the scene with z-buffering using the face index as their colors, and assign to the color image the generated images in the corresponding view.

3. For each face we project its vertices using the model-view and projection matrix of each view. Then we iterate over the color image pixels in the bounding box of the project face until its end or find a pixel with the same color of the face.

4. If the color of the face is the present in the bounding box, we save the following information:

   - **View**: A pointer to the view (camera and image information) from which the face is visible.
   - **Texture coordinates**: The coordinates of the projected vertices of the face through the view camera.
   - **Angle**: the angle of acquisition (computed from the face normal and the view direction).

Although this implementation is very efficient, its accuracy depends on the viewport resolution of the face size. Due to this fact, multiple faces could be projected onto the same pixel (very small faces or faces that are on a surface nearly parallel to the view direction). To solve this problem we adopted one of the solutions given by Callieri *et al.* [7]: in each pass render only the faces not yet detected as visible, until all faces are detected as visible. Algorithm 2 contains the pseudocode of the method.

As consequence, for each face we know the cameras from which it can be seen. Using this information, we can select the most orthogonal camera (best camera) for each face. Figure 3.1 shows a model obtained from 3 images/cameras, whose faces were colored with their most orthogonal cameras.

Algorithm 2 visibilityCalculation()

```
F ← set of faces of the mesh
V ← set of views
ni ← number of invisible faces
ni ← MAXINT
while ni > 0 do
  for each face f in F do
    for each view v in V do
      if f is not visible from v.camera then
        b ← f bounding box when projected onto v.image using v.camera
        for each pixel p in b do
          if p.color = f.color then
            f.addView(v)
          end if
        end for
      end if
    end for
    update ni
  end for
end while
```

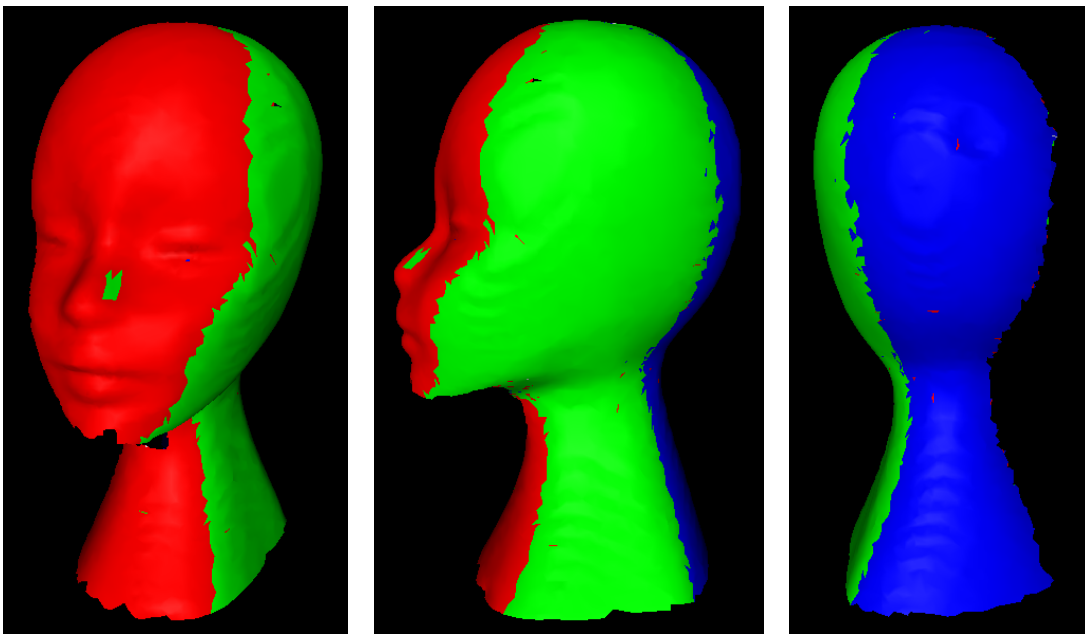
3.3 Atlas Generation

In this section we will describe how we can construct a projective texture atlas from a set of images in two parts. First we will explain how to partition the mesh into a set of patches such that the mapping distortion of each chart is minimized. After that we will describe how to map the charts to the discrete image grid.

In seeking a good texture atlas, the primary objective is to minimize distortion, such that large texture distances are not mapped onto small surface distances. One strategy is to define an energy functional for the mapping, and to try too minimize it. In our case, we adopt the atlas mapping distortion as this energy functional, and a heuristic in order to not produce too many charts. This functional is calculated from a distortion-based metric, a variation of the $\mathcal{L}^{2,1}$ metric [9], which allow us to "score" a partition in terms of mapping distortion. Given a chart C_i and its associated camera c_i, with direction n_i, the distortion metric $\mathcal{D}$ is then:

$$\mathcal{D}(C_i, c_i) = \iint_{x \in C_i} \|n(x) + n_i\|^2 dx \tag{3.1}$$

In Section 3.3.1 we describe our variational approach to construct an atlas structure for this problem and in Section 3.3.2 we detail how the atlas is parametrized.



(a) (b) (c)

Figure 3.1: Faces of a 3D model (with 10K faces) colored with their best cameras. Camera positioned at (a) camera 1 (red camera), (b) camera 2 (green camera) and (c) camera 3 (blue camera).

3.3.1 Distortion-Based Mesh Partitioning Algorithm

In the problem of constructing a good texture atlas from a set of images we have to give attention to two problems: minimize the mapping distortion, in order to reduce texture stretch, and minimize the number of charts, in order to have as large as possible continuous surface areas which are assigned to the same image.

Mapping distortion, in our problem, can be thought as the sum of the distortion errors $\mathcal{D}$ for all charts of the atlas. In order to reduce texture distortion in the texture-to-surface mapping (i.e. minimizes texture stretch, such that small texture distances are mapped onto large surface distances), we have to select, for each face, the best camera (from the set of cameras, the most orthogonal to the face, the one with the most accurate color information) to construct the texture map.

The second problem has to be attacked because an excessive number of charts will cause problems related to color discontinuity between different parts of texture, besides increasing the texture map occupied space.

Considering this tradeoff between the number of charts and mapping distortion, we develop a method for mesh partitioning, described in Algorithm 3. This algorithm, which is subdivided in two phases, tries to minimize the texture distortion, until no more patches could be created due to some heuristics.

In the next sections we will explain each part of the algorithm.

Algorithm 3 meshPartitioning()

```
np ← number of patches
np ← number of cameras
npb ← number of patches before adding a new patch
npb ← 0
patchGrowing()
while npb < np do
  npb ← np
  np ← patchAdding()
  patchGrowing()
end while
patchMerging()
```

Patch Adding

In order to bootstrap the process we select the low-distorted face to each camera to be a seed for the *patch growing* process, based on the distortion-based metric:

$$\mathcal{D}(f, c) = \|n_f + n_c\|^2 \cdot a_f \tag{3.2}$$

where c represents the camera (with directon n_c) and f the face (with normal n_f and area a_f).

For each seed selected we create a patch. After performing the *patch growing* step, every face of the mesh belongs to some patch. We will add a new patch to the atlas using the face with the biggest distortion error $\mathcal{D}(f_i, c_i)$ (Equation 3.2) as seed, what is in spirit a farthest-point heuristic. We add patches until we cannot select a face whose neighbors best cameras (those with lower-distortion in respect to the neighbors) are the same of the face, in order to minimize the number of charts and, consequently, to not produce too small ones. This method is described in Algorithm 4.

Patch Growing

The problem of patch growing can be stated as follows:

Given n seeds representing the n patches, assign each face to a patch.

From the previous section, we have this n seeds representing the n patches. We partition the elements by growing all the patches simultaneously using the flooding algorithm proposed by Cohen-Steiner *et al.* [9] and the $\mathcal{D}$ metric. Just like the Lloyd's algorithm, we want to cluster faces with low distortion $\mathcal{D}(f, c)$, in order to obtain a better partition (i.e. a partition with distortion smaller than the previous partition, what shows the variational nature of the method). The flooding algorithm proposed by Cohen-Steiner *et al.* [9], with some modifications, is described as follows:

1. For each seed face f of the patch P, we insert its three adjacent faces in a priority queue (f_1, f_2 and f_3), with priority equal to the distortion error between these faces

Algorithm 4 patchAdding()

$F \leftarrow$ set of faces of the mesh
$s \leftarrow$ new seed face
$s \leftarrow \oslash$
$e \leftarrow$ distortion error
$e_{max} \leftarrow$ max distortion error
$e_{max} \leftarrow 0$
for each face f in F **do**
  $e \leftarrow \mathcal{D}(f, f.patchObject.camera)$
  if $e > e_{max}$ **then**
    $e_{max} \leftarrow e$
    $n \leftarrow$ numbers of neighbors faces to f with same best camera
    $n \leftarrow 0$
    for each neighbor face f_n to f **do**
      if $f.bestCamera = f_n.bestCamera$ **then**
        $n \leftarrow n + 1$
      end if
    end for
    if $n = 3$ **then**
      $s \leftarrow f$
    end if
  end if
end for
if $s \neq \oslash$ **then**
  add a new patch with s as seed
end if
return number of patches

and P ($\mathcal{D}(f_1, P.camera)$, $\mathcal{D}(f_2, P.camera)$ and $\mathcal{D}(f_3, P.camera)$), and a pointer to the patch they are being tested against (P, in this case).

2. With these $3 \cdot number\ of\ patches$ in the queue, we perform the region-growing process as follows:

   (a) Pop out a face f from the priority queue (the face with smallest distortion will be popped out).

   (b) Check f patch assignment. If it has been assigned to a patch, do nothing and repeat (a).

   (c) Otherwise, assign f to the patch indicated by the pointer (P), insert its adjacent faces that were not assigned to a patch in the queue, with priority equal to the distortion error between the adjacent faces and the patch, and a pointer to the patch P, the same of f.

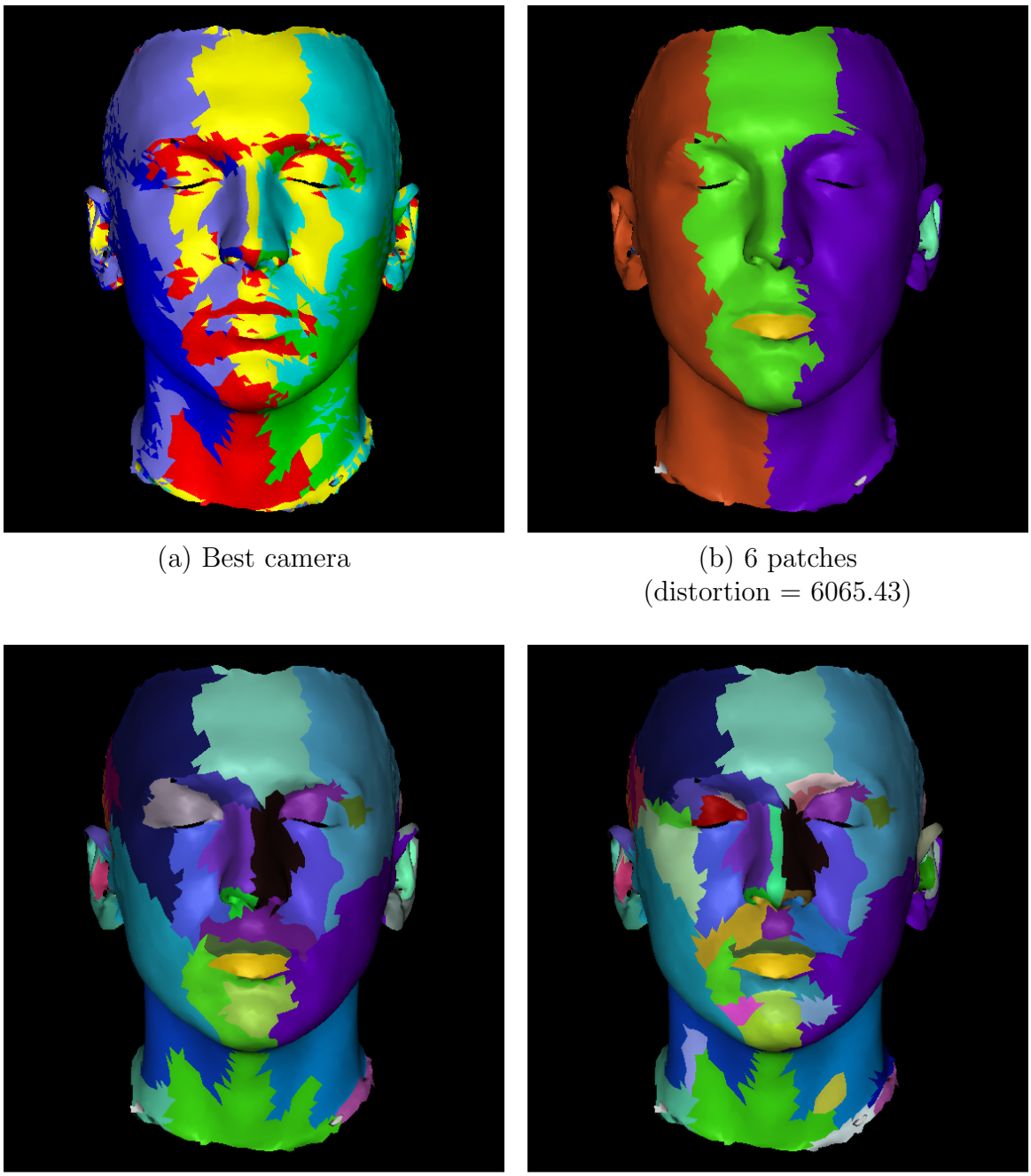
   (d) Repeat (a), (b) and (c) until the queue is empty, what means that each face of the mesh has been assigned to a patch.

The pseudocode to this algorithm is described in Algorithm 5.

Algorithm 5 patchGrowing()

```
Q ← priority queue
P ← list of patches
for each patch p in P do
  f ← p.seedFace
  for each neighbor face f_n to f do
    Q.push(p, f_n, D(f_n, p.camera)) {Equation 3.2}
  end for
end for
while Q ≠ ∅ do
  (p, f) ← Q.pop()
  if f.patchObject = ∅ then
    p.faceList.add(f)
    f.patchObject ← p
    for each neighbor face f_n to f do
      Q.push(p, f_n, D(f_n, p.camera)) {Equation 3.2}
    end for
  end if
end while
```

We repeat alternately the phases *patchAdding*() (Algorithm 4) and *patchGrowing*() (Algorithm 5) of the Algorithm 3 until we cannot add a new patch to the partition (from the restrictions that we have imposed). An example of this process is showed in Figure 3.2, in which the model was obtained from 6 images/cameras.



(a) Best camera

(b) 6 patches
(distortion = 6065.43)

(c) 30 patches
(distortion = 4859.82)

(d) 59 patches
(distortion = 4680.54)

Figure 3.2: A 3D model (with 10K faces) constructed from 6 images, whose faces in (a) were colored with their best cameras. Initially we grow a patch around the most orthogonal face to each camera (b). We iterate the algorithm (c) until we cannot add more patches (d). Note the decreasing of the distortion (calculated from Equation 2.4).

Patch Merging

Even so we worried about the tradeoff between the number of charts and mapping distortion in the patch adding/growing process, still it is possible to reduce the number of patches, without increasing the mapping distortion.

This reduction is possible because the resulting partition may contains adjacent patches with the same view (camera/image information). Since two adjacent patches with the same view have the same projective mapping, the distortion in the texture-to-surface-mapping after a patch merge operation will be the same as before.

Following this idea we develop a method to reduce the number of patches (and consequently the texture map size and color discontinuity) base on the adjacency information of each patch. The algorithm is described in Algorithm 7.

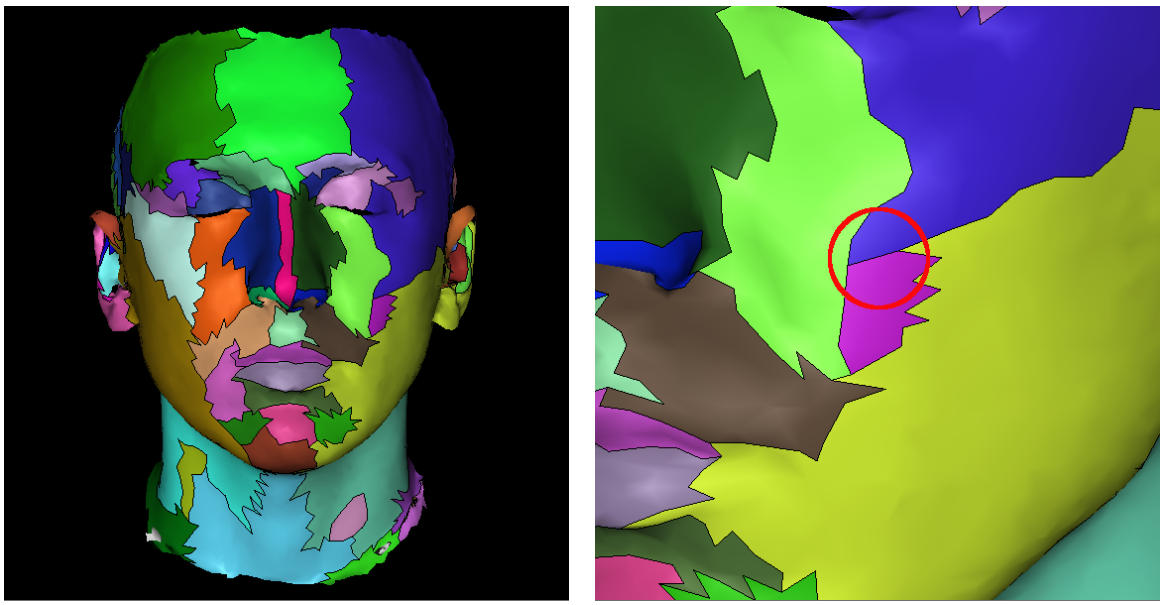
Algorithm 6 patchMerge(p_1, p_2)

for each face f in p_2 **do**
  $p_1.faceList.add(f)$
  $f.patchObject \leftarrow p_1$
end for
$P.remove(p_2)$

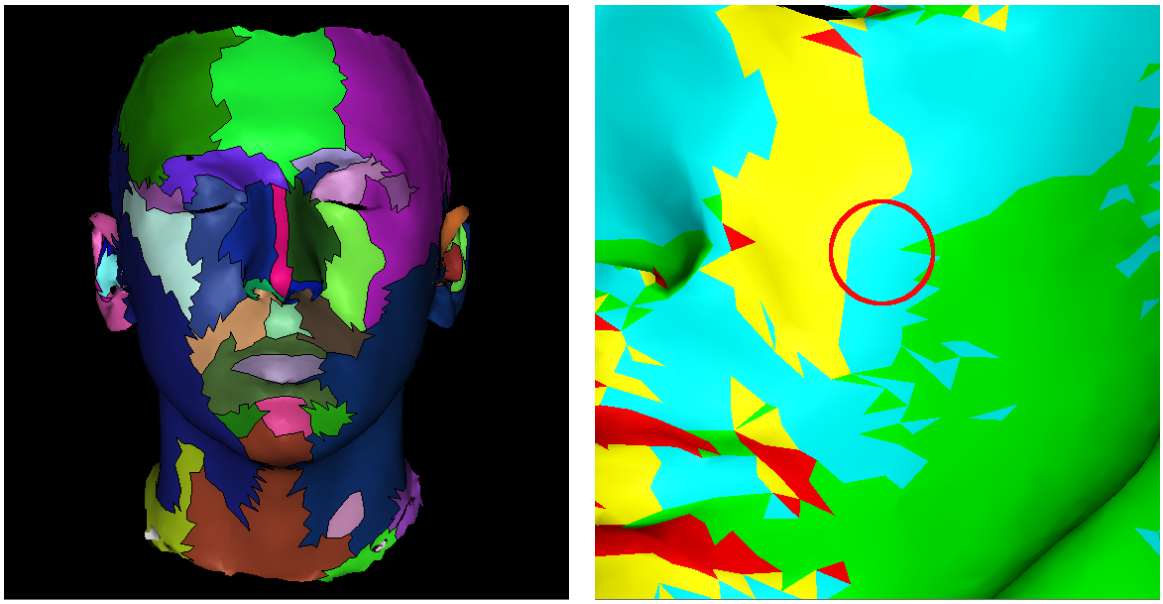
Algorithm 7 patchMerging()

$P \leftarrow$ list of patches
for each patch p in P **do**
  for each neighbor patch p_n to p **do**
    if $p.view = p_n.view$ **then**
      $patchMerge(p, p_n)$ {Algorithm 6}
      update patch neighbors
    end if
  end for
end for

Figure 3.3 shows the effects of the merging operation.



(a) (b)



(d) (c)

Figure 3.3: In (a) we show a partition obtained, with 59 patches. Note that the adjacent patches marked in (b) share the same camera (c), so they have to be merged. After merging all patches that rely on this case we have a new partition (d), with 39 patches.

3.3.2 Parametrization and Discretization

From the previous sections we have concluded the first step in the texture atlas construction process: partition the mesh in small number of patches, based on the distortion in the

texture-to-surface-mapping, such that each patch is associated to a image that contains the most accurate color information for that patch.

As we have explained in Section 2.2.2, the second step (chart parametrization) is not a problem for us, since the underlying parametrization of each patch is the inverse of the projective mapping of the patch camera. Hence, each projected patch is used as the parametrization domain for the surface patch. The mapping between each parametrization domain and surface domain is defined using discrete sets of values: the projected patch and mesh coordinates. This discretization is done by projecting each patch boundary onto its associated input image, using the parameters of its associated camera.

Is this way, we have partitioned our mesh into a set of patches such that each patch is associated to a region of an input image (obtained by projecting the 3D points of the patch vertices onto the image using the associated camera parameters) through a parametrization that is an inverse of a projective mapping, i.e, we have constructed a texture atlas. Figure 3.4 shows an example of mapping a texture atlas on a 3D model.

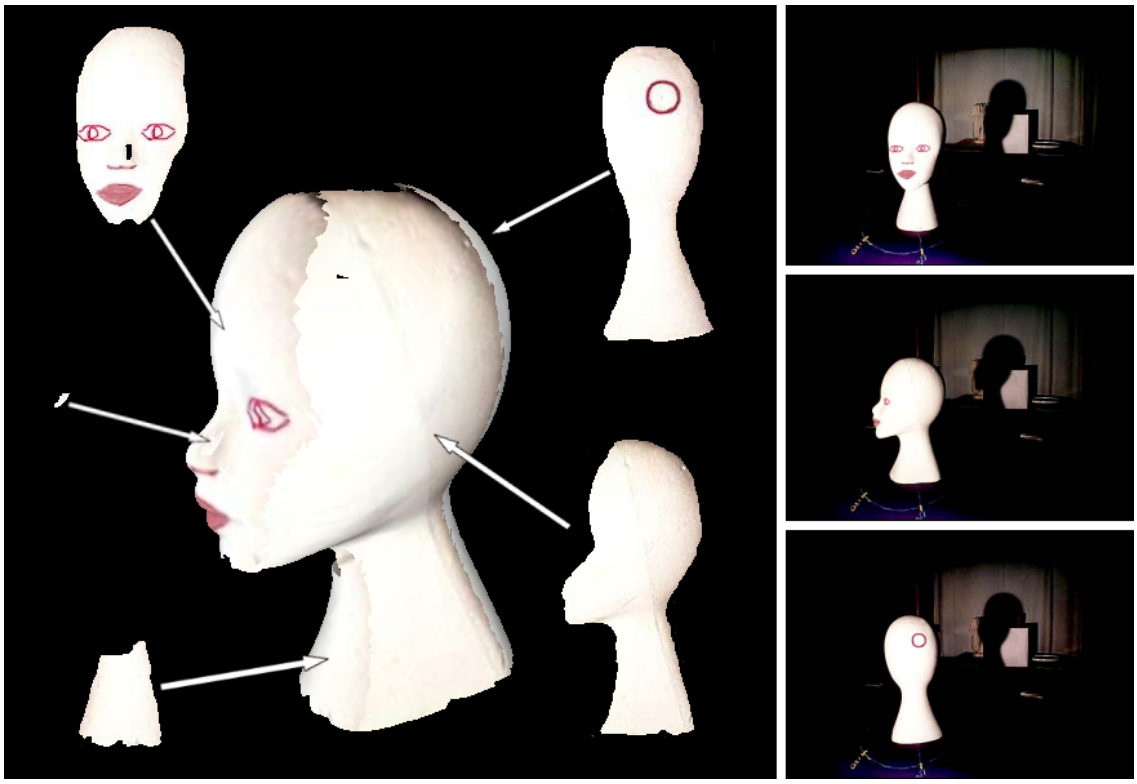


Figure 3.4: Mapping a texture constructed from a set of 3 images (on the right) on a 3D model (with 10K faces). Each patch (in this case, 5) is associated to a region of an input image (on the left).

In the next sections we will describe the optimizations developed for this pipeline (texture compression (Section 3.4) and blending (Section 3.6)) and how we pack our charts (Section 3.5) in a texture map.

3.4 Texture Frequency Analysis

Image resolution can be understood as its capacity to discriminate information of a given object. Although an image has a fixed resolution, the object texture contains different levels of detail (or frequencies). For this reason, an image has the potential disadvantage of using space inefficiently. Portions comprising low spatial frequency content require as much space as those with much higher spatial frequencies. As an example, faces of humans need high resolution details in areas such as eyes and mouth, while the appearance of other regions can be captured with fewer pixels per unit area.

This wasted space may not matter to applications which rely primarily on small, repeated textures to enhance the appearance of a simple environment. However, the production of realistic environments often requires the use of large textures which are uniquely applied to individual surfaces.

The texture frequency analysis for space optimization was explored in several works. In Hunter and Cohen [14] a frequency map for the image is computed using Fourier analysis. They partition the image based on this analysis to equalize the frequency content of each partition. Although this method is effective and simple, the resulting optimized image contains discontinuities. This idea is better explored in Balmelli *et al.* [2], in which the Fourier analysis is replaced by a wavelet analysis. From this analysis, a warping function is generated to uniformly distribute frequency content of the image.

Motivated by these works and by Carr and Hart [8], in which a multiresolution texture atlas is constructed from a texture frequency analysis, we develop a method to construct a multiresolution texture map based on frequency analysis using a Laplacian pyramid.

3.4.1 The Laplacian Pyramid

The Laplacian pyramid has been developed by Burt and Adelson [6] in order to compress images. It is basically a decomposition of the original image into a hierarchy of images such that each level corresponds to a different band of image frequencies.

The first step for generating a Laplacian pyramid is to low-pass filter the original image G_0 to obtain G_1 using a gaussian filter. G_1 is a reduced version of G_0 because the resolution was decreased. We then low-pass filter G_1 to obtain G_2 using the same filter, and so on. This sequence of images G_0, G_0, ..., G_n (where n is the number of levels of the pyramid) is called the Gaussian pyramid of G_0.

Each level of the Gaussian pyramid is obtained using the operation *REDUCE*:

$$G_l = REDUCE[G_{l-1}] \tag{3.3}$$

which means,

$$G_l(i,j) = \sum_{m=-2}^{2} \sum_{n=-2}^{2} w(m,n) G_{l-1}(2i+m, 2j+n) \tag{3.4}$$

A 5×5 Gaussian mask w (convolution kernel) is used to generate each pyramid level. When we apply a Gaussian filter on an image, the high-frequency components are discarded.

One way to save and separate these components is to subtract the Gaussian image from the original. This is called the Laplacian pyramid of an image.

To obtain the Laplacian pyramid, each image of a level is subtracted from the image of the level above. Due to the different image sizes, it is necessary to create interpolated images before this subtraction. This interpolation can be done in the process called *EXPAND*, the inverse of *REDUCE*. Being $G_{l,k}$ a image obtained when we expand G_l k times:

$$G_{l,0} = G_l \tag{3.5}$$

and, for $k > 0$,

$$G_{l,k} = EXPAND[G_{l,k-1}] \tag{3.6}$$

which means,

$$G_{l,k}(i,j) = 4 \sum_{m=-2}^{2} \sum_{n=-2}^{2} w(m,n) G_{l,k-1}(\frac{i-m}{2}, \frac{j-n}{2}) \tag{3.7}$$

Only terms for which $(im)/2$ and $(jn)/2$ are integers are included in this sum.

The Laplacian pyramid is a sequence of error images L_0, L_0, ..., L_n . Each one is the difference between two levels of the Gaussian pyramid. Thus, for $0 < l < n$,

$$L_l = G_l - EXPAND[G_{l+1}] = G_l - G_{l+1,1} \tag{3.8}$$

The original image can be recovered exactly by expanding and summing all the levels of the Laplacian pyramid, what means that the laplacian pyramid breaks up an image into components based on spatial frequency. The top level of the pyramid will contain the highest spatial frequency components, i.e, the edgiest of the edges. The bottom level will contain the lowest spatial frequency components. The intermediate levels contain features gradually decreasing in spatial frequency from high to low.

3.4.2 Assigning Frequencies to Faces

If we want to eliminate the wasted space caused by low spatial frequency regions in our texture map, we have to analyze the frequency content of each chart of our atlas and to partition them in regions of similar detail information.

In Section 3.3 we use a projective mapping distortion metric to create the charts. Now, since we are interested in splitting our patches based on a frequency content metric, we have to add a new attribute to the faces: its frequency content (level in the Laplacian pyramid).

The first level of the Laplacian pyramid contains the most high frequency regions of the input image, so if the texture region of a face has a very sharp edge, we will assign this level to the face. Naturally we will assign lower levels to faces whose texture region have little detail information. But how to decide which level?

Following Mallat and Zhong [20], in which Mallat has proven that the local image regularity is characterized by the decay of the wavelet transform amplitude across scales, we have a way to decide which level of the Laplacian pyramid assign to each face.

For each face of the mesh, we project its coordinates onto the images of different levels of the Laplacian pyramid (constructed from the input image linked with the patch), using the camera parameters of the associated patch. For each projected region, we compute the "detail content" through the function described in Algorithm 8

Algorithm 8 faceDetail($face$, $pyramidLevel$)

$factor \leftarrow 2^{pyramidLevel}$
$b \leftarrow face$ bounding box when projected onto $face.patchObject.view.image$
$b \leftarrow (b - 0.5 \cdot (factor - 1))/factor$ {triangle area coherence across scales}
$v1 \leftarrow face.v1$ projection onto $face.patchObject.view.image$
$v2 \leftarrow face.v2$ projection onto $face.patchObject.view.image$
$v3 \leftarrow face.v3$ projection onto $face.patchObject.view.image$
$v1 \leftarrow (v1 - 0.5 \cdot (factor - 1))/factor$
$v2 \leftarrow (v2 - 0.5 \cdot (factor - 1))/factor$
$v3 \leftarrow (v3 - 0.5 \cdot (factor - 1))/factor$
$im \leftarrow face.patchObject.view.laplacianImage(pyramidLevel)$
$l \leftarrow$ list of "detail content" of the face
for each pixel p in b **do**
  if $pointInsideTriangle(p, v1, v2, v3)$ **then**
    $l.add(im.getLuminance(p))$
  end if
end for
return MAX_VALUE(l)

A decay of the detail content between the level $k-1$ and k means that from the level k we are losing frequency content of the projected region. Therefore the Gaussian image corresponding to level $k-1$ of our Gaussian pyramid is the last image that preserves the detail content of the face projected region. In this way, the level that we have to associate to that face is level $k-1$. The method can be seen in Algorithm 9.

Figure 3.5 shows a example of running this algorithm on two models.

From this we assign a pyramid level to each patch. Given a patch P, its pyramid level L_P is the integer value closest to the area-weighted average of the face's levels (l is the level of a face f and s is its area):

$$L_P \simeq \frac{\sum_{f_i \in P} s_i \cdot l_i}{\sum_{f_i \in P} s_i} \tag{3.9}$$

Algorithm 9 assignFrequenciesToFaces()

```
F ← set of faces of the mesh
nl ← numbers of levels of the Laplacian pyramid
for each face f in F do
  d_f ← detail content of f
  d_f ← 0
  daux_f ← detail content of f
  daux_f ← 0
  for i = 1 to nl do
    daux_f ← faceDetail(f, i) {Algorithm 8}
    if daux_f < d_f,{a decay of the detail content} then
      f.level ← i
      break
    end if
    d_f ← daux_f
  end for
end for
```

3.4.3 Frequency-Based Mesh Partitioning Algorithm

After Section 3.3 we had a atlas structure such that each chart of the atlas is associated to a section of one input image, in a way that the texture map distortion is minimized.

Given the fundamentals of the previous section, we develop an algorithm to partition the charts based on the frequency content of the their faces. Our atlas structure is then modified: each chart of the atlas is associated to a section and a pyramid level of one input image. However, the atlas distortion does not increase or decrease, since in this process we do not change the face-to-camera mapping.

We apply the same algorithm developed in Section 3.3.1 with two modifications:

- Instead of a mapping distortion-based metric, we now use a texture frequency-based metric.
- We apply the process in each patch separately, since the results obtained when using a mapping distortion based metric are more relevant to our problem.

In this way, for each patch we create a set of "subpatches", clusters of faces with similar frequency content. This algorithm is showed in Algorithm 10.

From Figure 3.5 we can see that the noise of the input images is captured by Algorithm 9. To Algorithm 10 obtain better results, it is preferred to have as large as possible continuous surface areas with the same detail content. To achieve this, we adopt an iterative greedy approach: for each face, we assign to that face a higher detail pyramid level if that minimizes the number of different levels adjacent to that face.

Subpatch Adding

We will add subpatches based on the existence of groups of adjacent faces with the same pyramid level (detail content) through the use of a frequency-based metric:

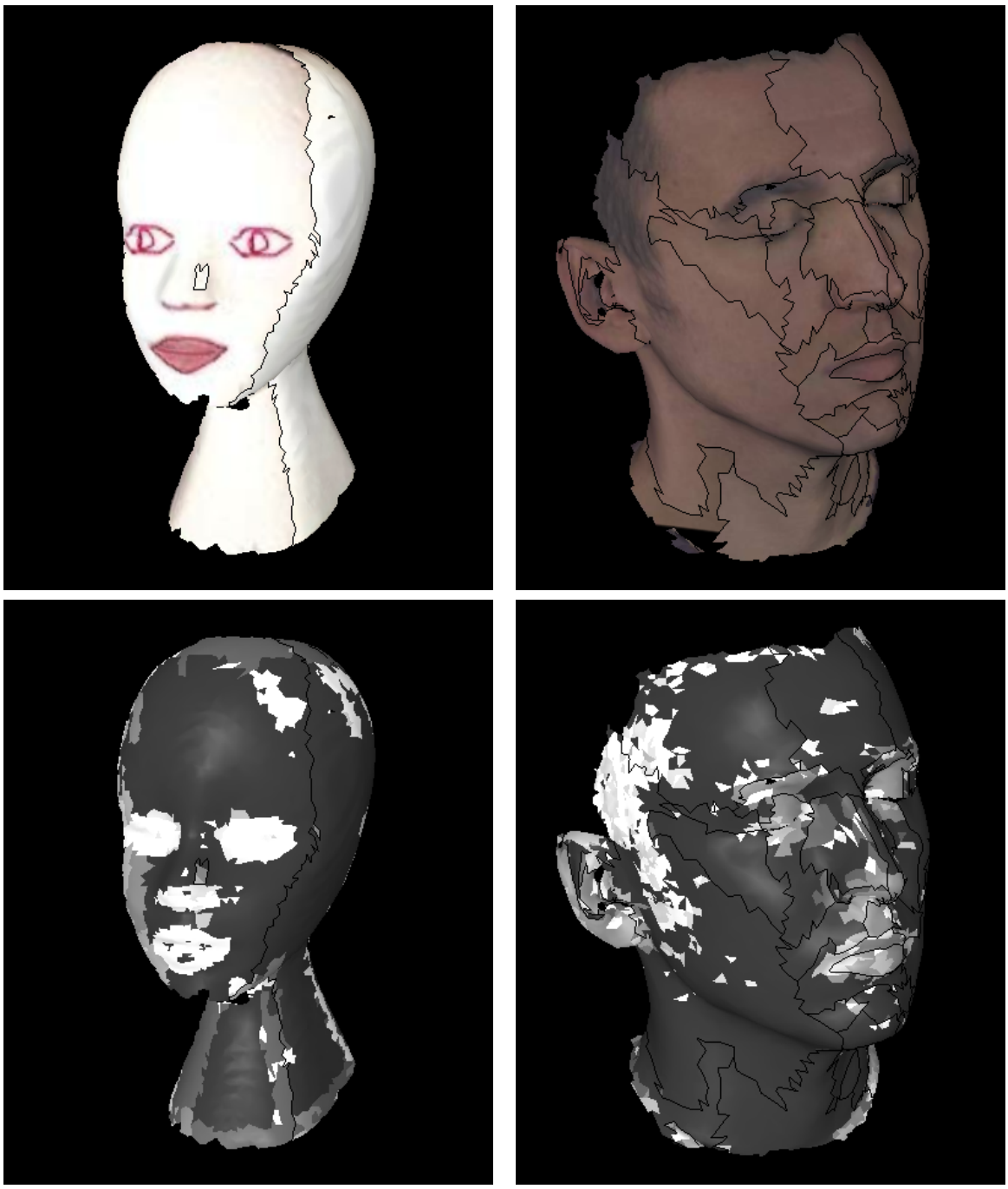


Figure 3.5: On the top two textured models with the patches frontier marked in black. On the bottom the pyramid level of their triangles (the faces are colored with a grayscale color, being that the color of the highest frequency pyramid is white and the color of the lowest frequency pyramid level is black). From the results we can see that the algorithm captures regions of fine detail (the eyes, nose and mouth, and some rugosities, of the model on the left, and the eyes, nose, mouth and the hair of the model on the right).

Algorithm 10 patchSplitting(p)

$np \leftarrow$ number of subpatches
$np \leftarrow 1$ {we create a subpatch with the same data of patch p}
while np is increasing **do**
  $np \leftarrow subpatchAdding(p)$
  $subpatchGrowing(p)$
end while
$subpatchMerging(p)$

$$\mathcal{F}(f, l) = \|l - l_f\| \cdot a_f \tag{3.10}$$

where l represents the level of the patch and f the face (with level l_f and area a_f). Following Algorithm 4, we will add a new subpatch using the face with the biggest frequency-based error $\mathcal{F}(f_i, l_i)$ (Equation 3.10) as seed, if the levels of the face neighbors are the same of the face. We describe this method in Algorithm 11.

Subpatch Growing

We use the same flooding algorithm for patch growing of Section 3.3.1, but with the frequency-based metric from Equation 3.10. In fact, for the growing process we have to add the term $MINFLOAT \cdot (l + l_f)$ to this metric. This is necessary because if a pyramid level k is assigned to a face f, and in the growing process this face is reached by the subpatch sp_1, pyramid level $k + 1$, and by the subpatch sp_2, pyramid level $k - 1$, the absolute difference between the levels of the patches and the face level is the same (1), but sp_2 has priority over sp_1, since it preserves the detail content of f

We detail this variation of Algorithm 5 in Algorithm 12.

The phases $subpatchAdding(p)$ and $subpatchGrowing(p)$ of the Algorithm 10 are repeated alternately using this frequency-based metric, until we cannot split more the patch. The result of this is a clustering of faces with similar frequency content.

Algorithm 11 subpatchAdding(p)

```
s ← new seed face
s ← ∅
e ← frequency error
e_max ← max frequency error
e_max ← 0
for each face f in p.subpatches do
  e ← F(f, f.patchObject.level)
  if e > e_max then
    e_max ← e
    n ← numbers of neighbors faces to f with same level
    n ← 0
    for each neighbor face f_n to f do
      if f.level = f_n.level then
        n ← n + 1
      end if
    end for
    if n = 3 then
      s ← f
    end if
  end if
end for
if s ≠ ∅ then
  add a new subpatch with s as seed
end if
return number of subpatches
```

Algorithm 12 subpatchGrowing(p)

```
Q ← priority queue
P ← list of subpatches
for each subpatch sp in p.subpatches do
  f ← sp.seedFace
  for each neighbor face f_n to f do
    Q.push(sp, f_n, F(f_n, sp.level)) {Equation 3.10}
  end for
end for
while Q ≠ ∅ do
  (sp, f) ← Q.pop()
  if f.subpatchObject = ∅ then
    sp.faceList.add(f)
    f.subpatchObject ← sp
    for each neighbor face f_n to f do
      Q.push(sp, f_n, F(f_n, sp.level)) {Equation 3.10}
    end for
  end if
end while
```

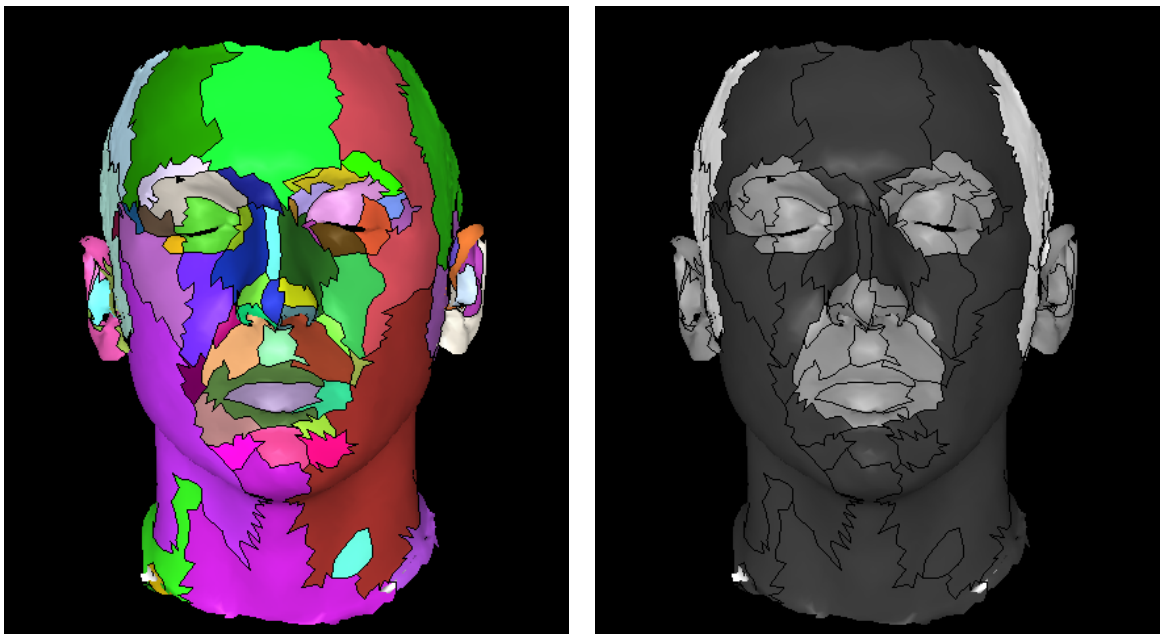
Subpatch Merging

At the end of the patch splitting process we have a similar problem to that found in Section 3.3.1: a splitted patch may contains adjacent subpatches with the same level in the pyramid. Since two subpatches with this peculiarity are linked to the same image in the pyramid, we should merge them in order to decrease the occupied space in the texture map. The algorithm, a variation of Algorithm 7, is described in Algorithm 13.

Algorithm 13 subpatchMerging(p)

```
for each subpatch sp in p do
  for each neighbor subpatch sp_n to sp do
    if sp.level = sp_n.level then
      patchMerge(sp, sp_n) {Algorithm 6}
      update subpatch neighbors
    end if
  end for
end for
```

In Figure 3.6 we show a example of the results of the frequency-based patch splitting process.



(a) (b)

Figure 3.6: Running the frequency-based patch splitting process on a model, originally partitioned in 39 charts, obtaining 70 subcharts (a), each one associated with a pyramid level (b). The hair have a detail content correspondent to the level 0 of the pyramid (highest frequencies); the ears, mouth, nose and eyes, level 1; and the remaining portion of the face level 3.

3.4.4 Re-discretization

As we saw in Section 3.3.2, the parametrization of each patch is the inverse of the projective mapping of the patch camera, and the discretization is done by projecting each patch boundary onto its associated input image, using the parameters of its associated camera.

Based on the results of the previous section we have to re-discretize our new patches (obtained from the subpatches created), since we have associated to each subpatch an image obtained from the gaussian pyramid of the input image associated to the original patch. This re-discretization is detailed in Algorithm 14.

Therefore we now have partitioned our mesh in a set of patches such that each patch is associated to a region and pyramid level of an input image through a parametrization that is a inverse of a projective mapping.

Algorithm 14 atlasRediscretization()

$P \leftarrow$ list of patches
for each patch p in P **do**
  for each subpatch sp in p **do**
    $factor \leftarrow 2^{sp.level}$
    $sp.image \leftarrow getGaussianImage(p.image, sp.level)$ {gaussian image at level $sp.level$ from the gaussian pyramid of $p.image$}
    $sp.bBox.x \leftarrow ((sp.bBox.x - 0.5 \cdot (factor - 1))/factor$
    $sp.bBox.y \leftarrow ((sp.bBox.y - 0.5 \cdot (factor - 1))/factor$
    $sp.bBox.width \leftarrow ((sp.bBox.width - 0.5 \cdot (factor - 1))/factor$
    $sp.bBox.height \leftarrow ((sp.bBox.height - 0.5 \cdot (factor - 1))/factor$
  end for
end for

3.5 Packing the Charts

At this point of the process we have a number of patches, each one linked to a section of one input image and a level of the image gaussian pyramid (texture patch) through a parametrization that is a inverse of a projective mapping. These texture patches need to be packed efficiently into a single texture map. But how?

As we have seen in Section 2.2.3, we adopt a heuristic similar to Sander *et al.* [24], but simpler. Basically, for each chart, we clip the bounding box of the texture patch (by projecting the 3D points of the patch vertices onto the image using the associated camera parameters). In fact we do not clip the exactly bounding box of the texture patch, but a bigger bounding box, to avoid rendering artifacts at chart boundaries, such as discontinuities. If the lower left coordinate of the texture patch bounding box is (x_l, y_b) and upper right coordinate is (x_r, y_t), we clip the texture patch bounding box defined by the coordinates $(x_l - 1, y_b - 1)$ and $(x_r + 1, y_t + 1)$.

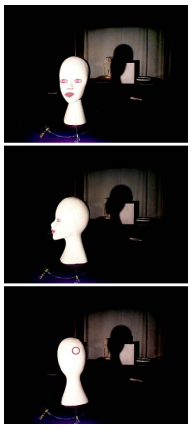
We then sort these clipped regions by height and, in order of decreasing height, place them sequentially into rows until a inserted region exceeds the texture map width (given by the user). If this happens, we place this region just above the first region of the previous row, until all texture patches have been inserted in the texture map.

The algorithm is described in Algorithm 15.

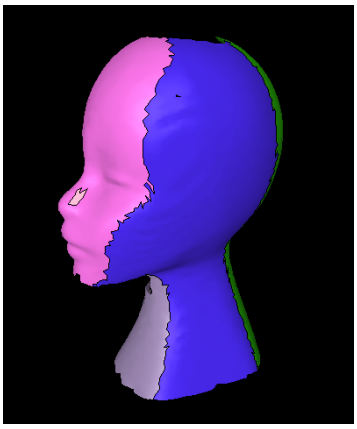
After this packing process we have a texture map. Therefore we have to update the parametrization of each chart, because the parametric domain have changed to the texture map image. This re-parametrization is done by simply verifying the correspondences between the projected patch onto the input image (and image level in the Gaussian pyramid) and the texture map image. Figure 3.7 shows an example the packing.

Algorithm 15 patchPacking($width$)

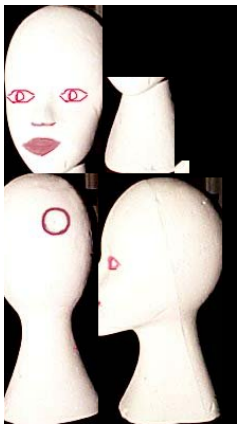
$P \leftarrow$ set of patches
$P_s \leftarrow$ set of patches, ordered by texture patch box height
$x \leftarrow$ left coordinate, in the texture map, of the next texture patch bounding box to be placed
$y \leftarrow$ lower coordinate, in the texture map, of the next texture patch bounding box to be placed
$p_p \leftarrow$ pointer to the last patch inserted
$p_f \leftarrow$ pointer to the first patch inserted in the previous line
$P_s \leftarrow sortPatchesByHeight()$
$x \leftarrow 0$
$y \leftarrow 0$
for each patch p in P_s **do**
  if p is the first patch of P_s **then**
    $p_p \leftarrow p$
    $p_f \leftarrow p$
  end if
  $p.bBoxTexture \leftarrow (x, y, p.bBox.width, p.bBox.height)$
  $placePatchOnTheTextureMap(p, x, y)$
  if $x + p.bBox.width > width$ **then**
    $x \leftarrow 0$
    $y \leftarrow y + p_f.bBox.height$
    $p_f \leftarrow p$
  else
    $x \leftarrow x + p_p.bBox.width$
  end if
  $p_p \leftarrow p$
end for



(a) Input images (3 images)

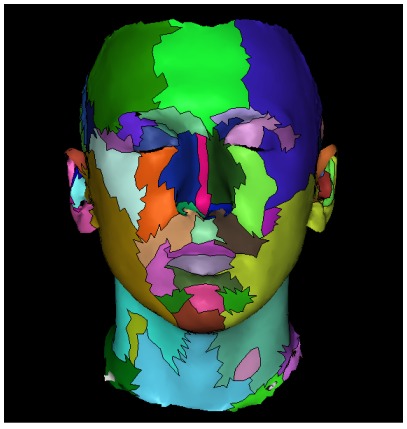


(b) Mesh partition (5 patches)

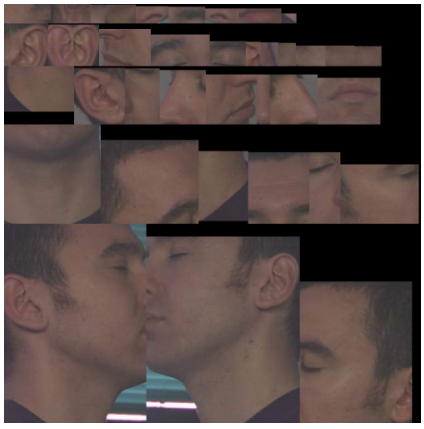


(c) Texture map (dimensions=220×396)

(a) Input images (6 images)



(b) Mesh partition (39 patches)



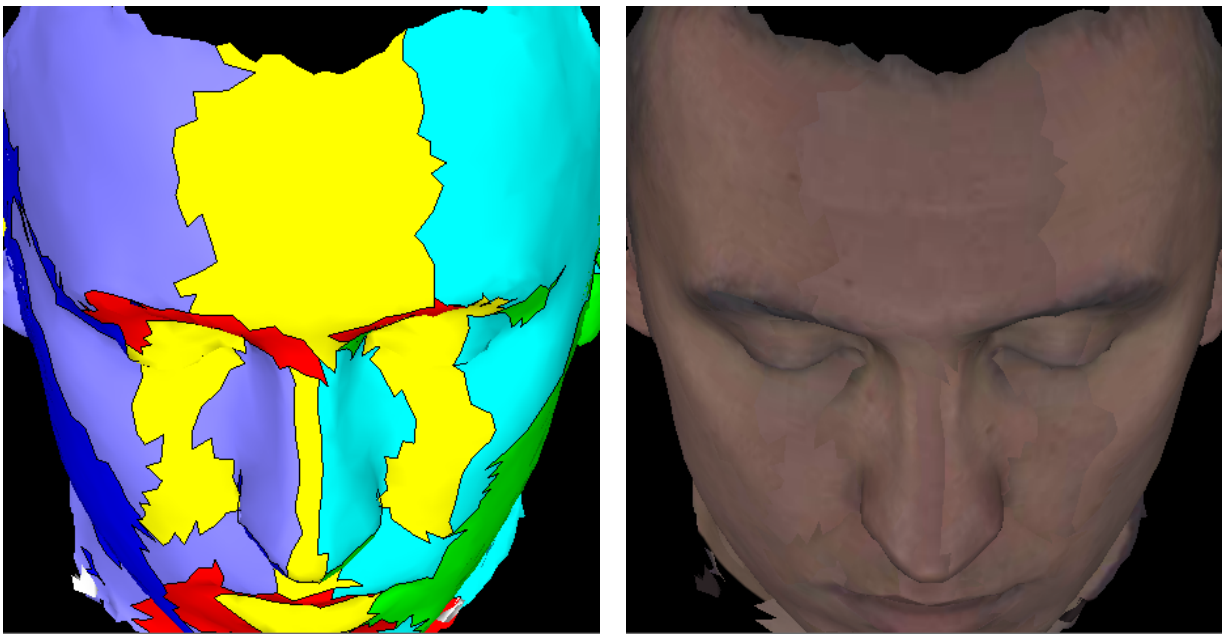
(c) Texture map (dimensions=750×755)

Figure 3.7: Results of the packing algorithm on two models. On the top a model constructed from 3 images (a), partitioned in 5 patches (b) and the texture map obtained by the packing algorithm (c). On the bottom a model constructed from 6 images (a), partitioned in 39 patches (b) and the texture map obtained by the packing algorithm (c).

3.6 Improving Continuity

One of the major problems when we construct a texture map using different input images is the color discontinuity between the parts of the mapped texture, as shown in Figure 3.8. This discontinuity is caused by different illumination conditions while capturing the images, since the pictures are taken by modifying the relative position between the camera/light source and the object.

Several approaches have been presented to reduce this effect. In our case the natural solution is to use the color and illumination difference between images assigned to adjacent patches. But how to combine this information in order to create smooth transitions between them?



(a) (b)

Figure 3.8: The results of our texture atlas construction applied on a 3D model. The model is colorized with the patch best camera id (a) and with the texture obtained from the input images (b). By looking to the forehead of the model (b) we clearly note that parts of three different input images were used in this region, and that the light source comes from the left side of the head.

A solution to this problem, as proposed by Callieri *et al.* [7], could be reached using the redundant information of the texture map. In their work the frontier faces (those that are on the frontier between adjacent patches) are represented in a redundant manner in the texture map, what means that if a face f is on the frontier between two patches p_1 and p_2, the texture map will store the color information of the face when using the mapping to image i_1 assigned to patch p_1 and the color information of the face when using the mapping to image i_2 assigned to patch p_2.

Although we do not represent any faces of our model in a redundant manner in the texture map (since our texture atlas is obtained from a partition of the model, and consequently, no face belongs to more than one patch), we know that two faces share a common edge. So, if a edge is shared by two frontier faces f_1 and f_2, which belongs to patch p_1 and p_2, respectively, this edge is mapped with color and illumination information of the face f_1 when using the mapping to image i_1 assigned to patch p_1 and color and illumination information of the face f_2 when using the mapping to image i_2 assigned to patch p_2.

Figure 3.9 illustrates, for the 1D case (axis x represents the pixels and axis y the colors), how this redundant information could be used to smooth the transition between two adjacent regions. In Figure 3.9.a we can see a example of the discontinuity of two functions, which happens in the point x_f. The point x_f, when assigned to the function f_1, has color c_1, and when assigned to the function f_2, has color c_2. Let $\overline{c}_{1,2}$ be the mean color of c_1 and c_2. So the difference $\overline{c}_{1,2} - c_1$ is how do we have to change the color of x_f when it is assigned to the

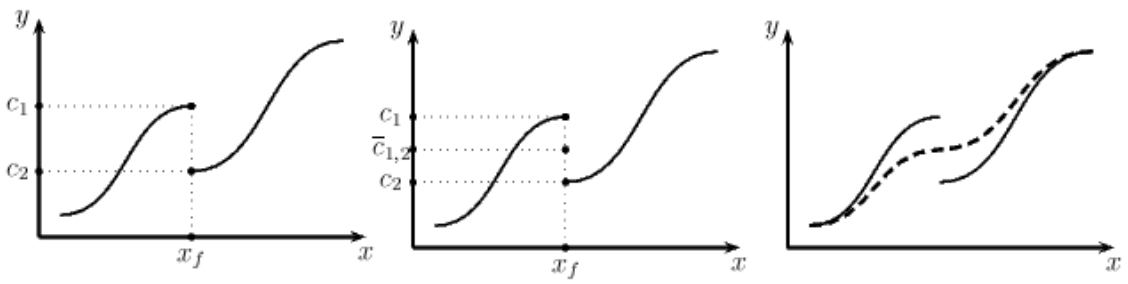


Figure 3.9: Smoothing the transition between two adjacent 1D functions.

function f_1 to became compatible with color c_2. In the same way, the difference $\overline{c}_{1,2} - c_2$ is how do we have to change the color of x_f when it is assigned to the function f_2 to became compatible with color c_1 (Figure 3.9.b). Once these "correction factors" have been calculated, we can propagate this difference in order to create a smooth transition between the functions (dashed line in Figure 3.9.c).

In our case, as mentioned before, each frontier edge is projected twice in the texture map. For each one of these edges we compute the color difference between corresponding texels in the two projected lines (correction factors), in order to create a signed RGB image (called "texture correction image"). These correction factors are calculated as exposed in Algorithm 17.

Algorithm 16 paintFrontierEdge(o_1, d_1, o_2, d_2)

$t_1 \leftarrow 0$
$t_2 \leftarrow 0$
$edgeSize_1 \leftarrow \|d_1 - o_1\|$
$edgeSize_2 \leftarrow \|d_2 - o_2\|$
while $t_1 \leq 1$ **do**
  $color_1 \leftarrow$ color of a pixel in the $edge_1$
  $color_2 \leftarrow$ color of a pixel in the $edge_2$
  $correctionColor \leftarrow$ correction color of a pixel in the $edge_1$
  $color_1 \leftarrow textureMap.getColor(o_1 + t_1 \cdot (d_1 - o_1))$
  $color_2 \leftarrow textureMap.getColor(o_2 + t_2 \cdot (d_2 - o_2))$
  $correctionColor \leftarrow \frac{color_1 + color_2}{2} - color_1$
  $colorCorrectionImage.setColor(o_1 + t_1 \cdot (d_1 - o_1), correctionColor)$
  $t_1 \leftarrow t_1 + 1/edgeSize_1$
  $t_2 \leftarrow t_2 + 1/edgeSize_2$
end while

This process applied on a atlas is shown in Figure 3.10.

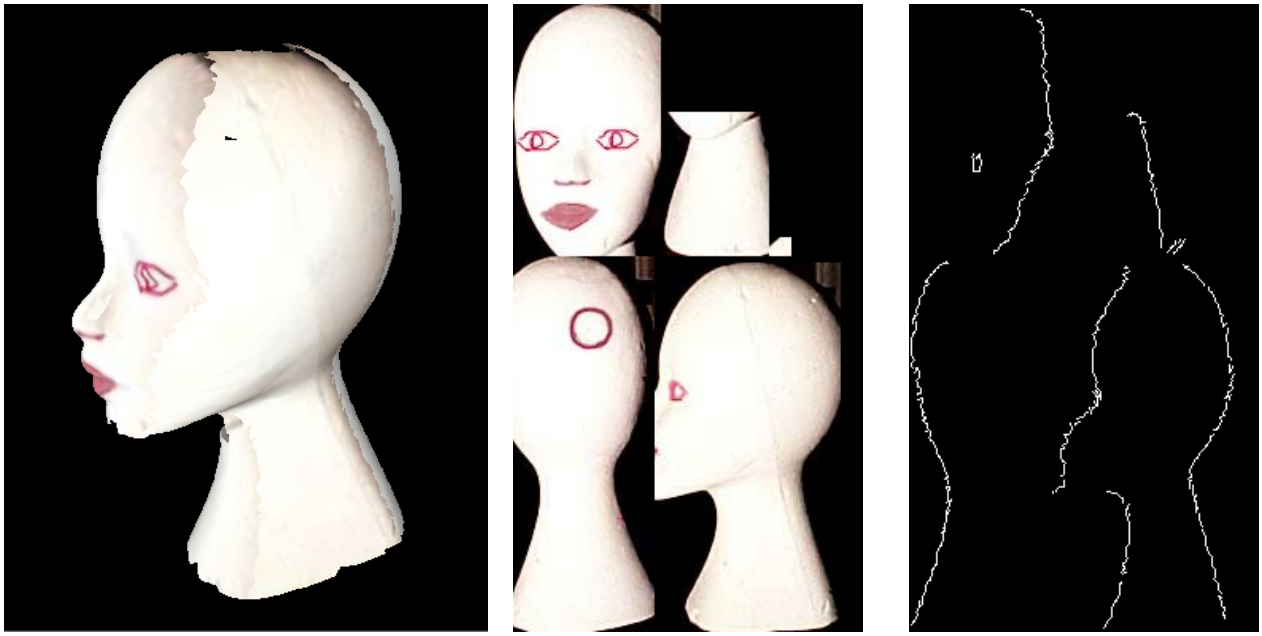
With these factors calculated we are able to perform a diffusion of them over the whole texture space. But how to do that in a efficiency and accurate way?

This is a classical problem of sparse interpolation, and we have studied two methods to solve it: Filtering and Diffusion.

Interpolation using filters is solved basically placing smoothing filters on the points to be

Algorithm 17 edgesDifference()

```
F ← set of frontier faces
for each face f in F do
  for each edge e of f do
    if e is a frontier edge then
      v2 ← first vertex of e
      v2 ← second vertex of e
      fn ← face that shares e with f
      Pf ← f associated patch
      Pfn ← fn associated patch
      o1 ← projection of v1 onto the texture map, through Pf camera
      d1 ← projection of v2 onto the texture map, through Pf camera
      o2 ← projection of v1 onto the texture map, through Pfn camera
      d2 ← projection of v2 onto the texture map, through Pfn camera
      paintFrontierEdge(o1, d1, o2, d2) {Algorithm 16}
    end if
  end for
end for
```



(a) Model with texture (5 charts)

(b) Texture map (dimensions=220×396)

(c) Texture correction image

Figure 3.10: The results when applying Algorithm 17 on a 3D model. In (a) we apply the texture on the model (note the discontinuities between adjacent patches). From this atlas we construct a texture map (b) and a texture correction image (c), which stores the correction factors for the frontier edges (showed in white color, for visualization purposes).

interpolated and convolving these filters. The Gaussian smoothing filter is a 2D convolution operator that is used to smooth images. In our case we want to use this operator only on the texels of the frontier edges of the texture correction image. Because these samples are not evenly spaced, this convolution cannot be done. The solution is to do a multiscale analysis using the wavelet transform [20]. A wavelet transform with a gaussian wavelet function is applied across each scale of the texture correction image. From the lower scale transformed image we compute the missing values compared to the higher scale transformed image, and add these values to the higher scale transformed image. This process is done until we get the first scale. The result is a smooth interpolation of the texels of the frontier edges of the texture correction image.

This idea was used by Callieri *et al.* [7], based on the pull-push interpolation method developed by Gortler *et al.* [12]. Based on the two dimensional scattered data approximation algorithm described by Burt and Adelson [6], which uses the low resolution data to fill in the void texels at higher resolutions.

Basically the method developed by Gortler *et al.* [12] has two phases:

- **Pull :** In this phase a sequence of lower resolution images is constructed from the texture correction image. Note that the size of the missing region is reduced in the reduced pyramid levels.

- **Push :** In this phase, information from each lower resolution grid is combined with the next higher resolution grid, filling in the void texels while not unduly blurring the higher resolution information already computed.

The process is iterated until all the texels of the texture correction image are filled.

We developed a method which produces smoother results, since it is not based on interpolation. We consider the sparse points (correction factors) as heat sources and solve the problem applying the heat (diffusion) equation on each heat source, which represents the flow of heat from that source.

This solution was also explored for constructing elevation models from level sets [11] and approximating lighting on 2D drawings [15]. The correction factors between frontier edges remain fixed; unknown values texels are relaxed across the image. Given a field of values P (texture correction image) to be interpolated, and a velocity field V, initially zero, each iteration is computed by:

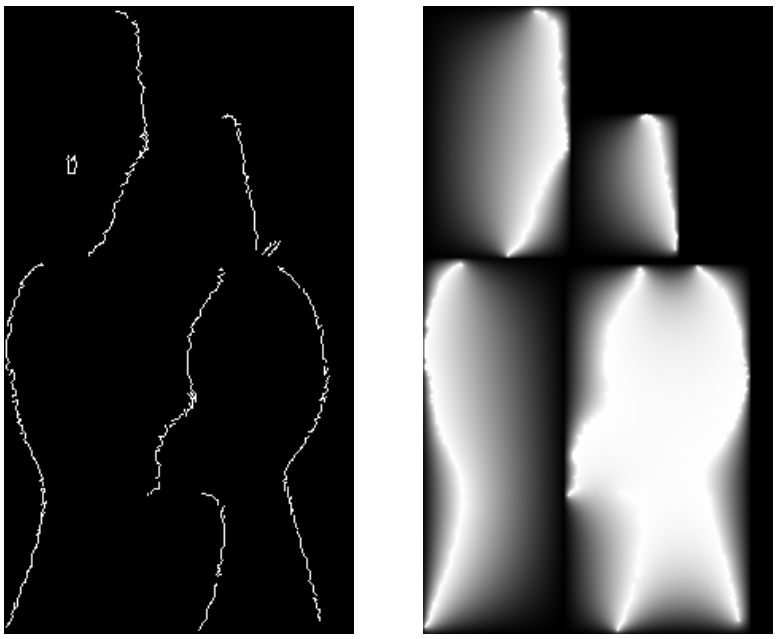
$$V'_{i,j} = d \cdot V_{i,j} + k \cdot (P_{i-1,j} + P_{i+1,j} + P_{i,j-1} + P_{i,j+1} + 4 \cdot P_{i,j}) \tag{3.11}$$

$$P_{i,j} = P_{i,j} + V'_{i,j} \tag{3.12}$$

As experienced by Johnston [15], values $d = 0.97$ and $k = 0.4375$ minimize the time of convergence. Iterations are run until the mean-squared velocity per pixel reaches a error tolerance (equilibrium temperature).

Figure 3.11 shows the results of such method executed on a texture correction image. Note that we use the diffusion equation on each chart separately, since we are only considering the influence of adjacent patches to smooth the interior area of a chart.

The problem of this method is that the diffusion equation takes a long time to converge. A solution to this efficiency problem is to explore the idea of the multiscale analysis explained above, or more specifically, a multigrid solver (detailed in the tutorial by Briggs [5]).



(a) Texture correction image (b) Smooth correction image

Figure 3.11: Applying the diffusion equation on the texture correction image (a), obtaining in this way a smooth correction image (b) (again, the correction factors are showed in white color, for visualization purposes).

Multigrid methods are very used to solve partial differential equations on a 2D grid. For this reason, it is totally suitable for our problem, since we want to solve a diffusion equation on a 2D grid. The method has two operators: restriction and prolongation, equivalents to the pull an push operators described before.

In multigrid computing, restriction and prolongation operators are used in any order of the process (Full Multigrid). Because of the simplicity of diffusion equation, we use these operators just like the pull and push operators by Gortler *et al.* [12]:

- **Restricting :** A sequence of lower resolution images is constructed from the texture correction image. Note that the size of the missing region is reduced in the reduced pyramid levels.
- **Prolongating :** Information from each lower resolution image, after applying the diffusion equation (Equations 3.11 and 3.12) until convergence, is combined with the next higher resolution image, filling in the void texels. Then we apply the diffusion equation until convergence on this higher resolution image.

This algorithm is detailed in Algorithm 18. As mentioned before we apply the diffusion equation on each chart separately, since we are only considering the influence of adjacent patches to smooth the interior area of a chart.

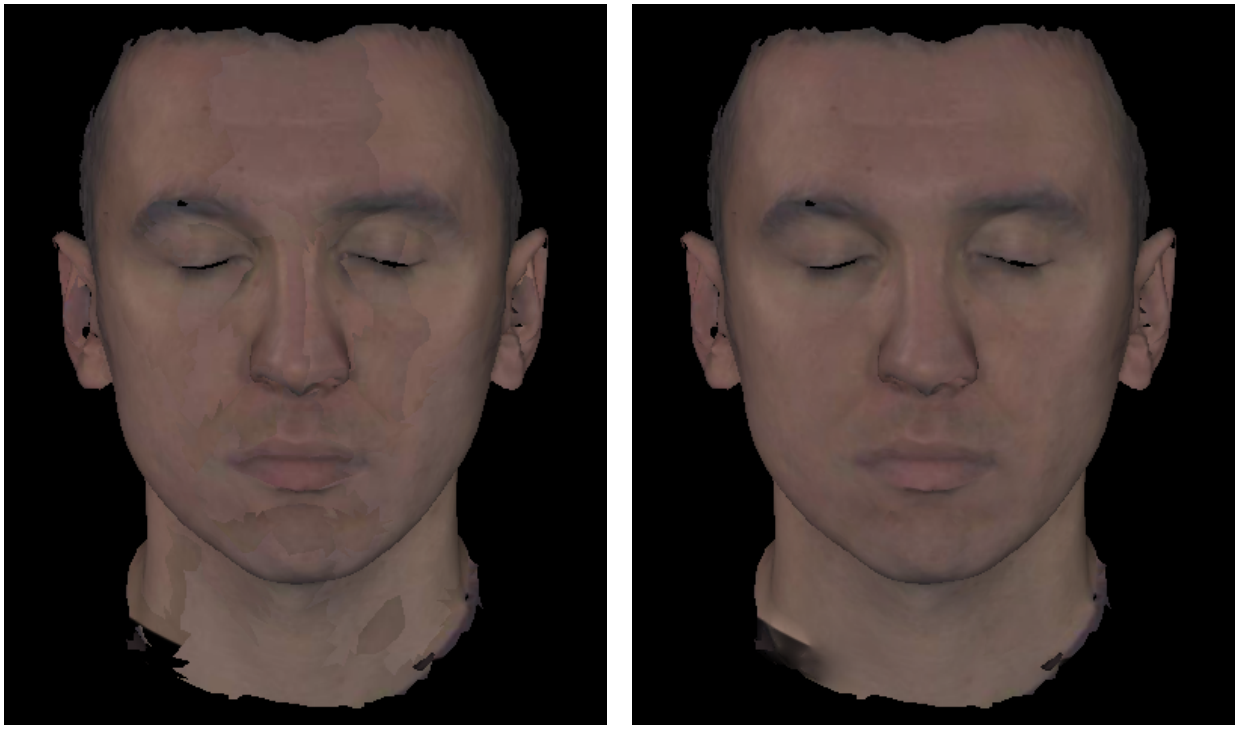
This method is very efficient since the time of convergence of the diffusion equation depends on the size of the missing region and the size of the image. The size of this missing

Algorithm 18 smoothChart(p)

$IR \leftarrow$ set of images generated in the restriction phase
$IP \leftarrow$ set of images generated in the prolongation phase
$IR_0 \leftarrow cropImage(p.bBoxTexture)$
$w \leftarrow p.bBoxTexture.width$
$h \leftarrow p.bBoxTexture.height$
$k \leftarrow 1$
while $w \geq 2$ and $h \geq 2$ **do**
  $IR_k \leftarrow downsample(IR_{k-1})$
  $w \leftarrow w/2$
  $h \leftarrow h/2$
  $k \leftarrow k + 1$
end while
for $l = k - 1$ to 0 do **do**
  $IP_l \leftarrow upsample(IP_{l+1})$
  $IP_l.copyFactors(IR_l)$
  $IP_l.applyDiffusionEquation()$
end for
return IP_0

region in the higher image, for example, is very small compared to if we did not have used the multigrid method.

After doing that, we add this smooth correction image to the previously calculated texture map, obtaining a uniform and continuous texture map (Figure 3.12).



(a) (b)

Figure 3.12: Mapping a texture on a 3D model without (a) and with (b) color continuity improvement.

Chapter 4

Attribute Editing

A attribute editing system makes it possible to enhance the visual appearance of a 3D model by interactively adding details to it (colors, normals, etc). 3D texture painting was first introduced by Hanrahan and Haeberli [13]. In this work they developed a surface painting system that allows the user to directly paint on 3D shapes (i.e. paint its vertices). However, in most cases, the desired precision for the colors is finer than the geometric details of the model. The solution to this problem is to use texture-mapping, but some problems could occur when using this technique:

- A parametrization of the surface (or inversely, a surface-to-texture mapping) is required, but finding a good surface parameterization is not trivial.
- The brush strokes are often distorted because of the underlying surface-to-texture mapping. If an area of the 3D surface is stretched out in the texture map, the brush stroke becomes small in the 3D view, and vice-versa. If an area of the 3D surface is split in the texture map, the discontinuity becomes visible in the painted model.

Based on these requirements (minimize texture stretch and provide a good parametrization of the surface), we conclude that projective texture atlas serves as the foundation for an attribute processing framework. In the case of visual properties, such as color, a texture atlas is an efficient color representation for painting systems.

Following these ideas we develop a texture painting/editing application that allows the user to directly paint on the charts of the atlas, and show several applications of this framework. In Section 4.1 we detail our texture painting/editing application. Section 4.2 explains how to construct good texture atlases (based on the ideas of Chapter 2) for the texture painting/editing framework.

4.1 User Interface

A 3D paint application allows the user to paint a model using all the standard painting and image manipulation tools. To make this possible, we develop an interactive and intelligent texture painting/editing application that allows the user to paint with different types of pigments and patterns.

The input of the application is a 3D mesh and a texture atlas for this mesh. The parametrization of this atlas is the inverse of the projective mapping from the camera associated to each chart (real cameras, as we have detailed in Chapter 2, or automatic generated virtual cameras, which construction will be explained in Section 4.2).

An important difference between our application and other existent 3D painting systems is that, instead of painting on the 3D model, the user paints directly on the charts of the atlas, which makes the painting process easier.

The interface has two views, a model view and a charts view (here we consider a view a window and a toolbar with different controls for that window). The model view contains the 3D model, the camera definitions and other features (Section 4.1.1). The charts view keeps the chart that is being edited (Section 4.1.2) and the brush options (Section 4.1.3).

Figure 4.1 is a screen dump showing the interface of the application, with the model view on the left and the charts view on the right, and the toolbar of each view on the bottom.

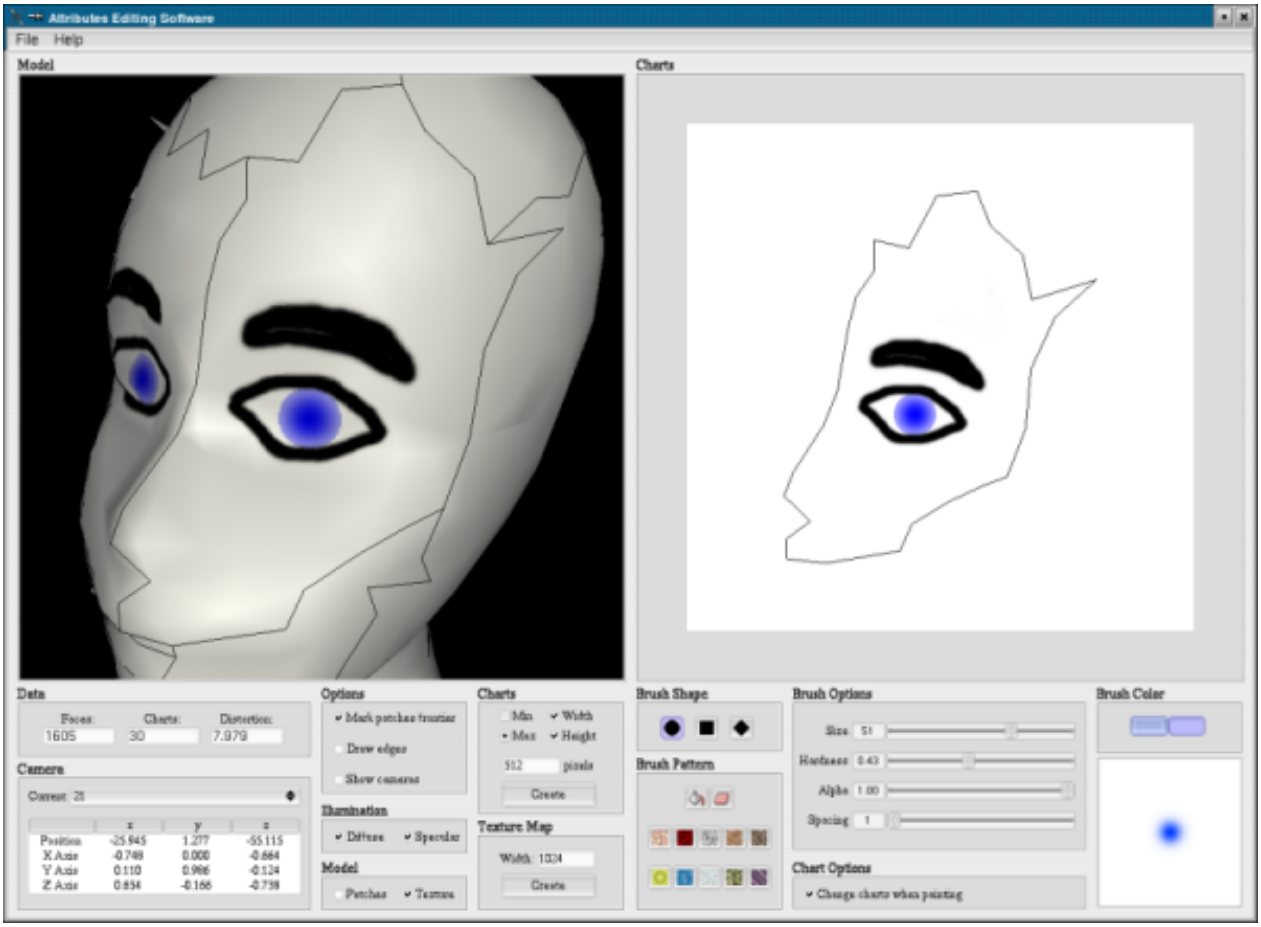


Figure 4.1: The attribute editing application interface.

4.1.1 The Model View

The model view can be separated in two parts: a OpenGL window which holds the 3D model and a toolbar with many controls related to the cameras, the model and the texture atlas.

The 3D model window has a powerful control to scale, translate and rotate the 3D model. It supports many options of visualization (view the painting results, mark the patches, mark the patches frontier, draw the face edges, show the cameras associated to each chart, etc). A screenshot of this window is showed in Figure 4.2.

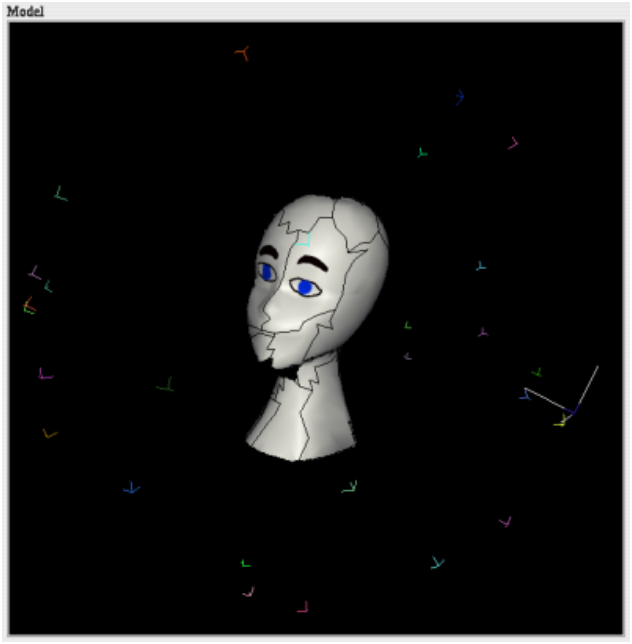


Figure 4.2: The 3D model window of the attribute editing application with the model being painted and the cameras of each chart.

When the user uses the mouse to scale, translate or rotate the model, the application looks for the camera closest to the OpenGL camera position, and loads the chart associated to this camera in the charts view.

The toolbar just above the 3D model window (Figure 4.3) has controls related to:

- **Charts camera:** The user can select the camera from which the model is being seen. When a camera is selected, the OpenGL updates its camera, using the chart camera parameters, for the 3D view, and the chart associated to that camera is loaded in a square OpenGL window (4.1.2), and the main camera parameters (view position and XYZ axis) are showed in a table.

- **Visualization options:** The 3D model can be visualized in many ways. The user can mark/not mark the patches on the model, view/not view the cameras of the atlas, draw/not draw the face edges and the patches frontier, change lighting conditions and map/not map the painted texture on the model.

- **Charts resolution:** The user can choose the resolution of the texture map. Although we already have a texture atlas and a parametrization of this atlas (inverse of projective mapping), we have not defined yet a discretization of each atlas in a texture map (except if the texture atlas has been produced from a set of images, as detailed in Chapter 3).

- **Texture map construction:** Based on an user-defined texture map width, the application packs the charts, accordinly to the pack algorithm described in Section 3.5 (except if the texture atlas has been constructed from a set of images, as detailed in Chapter 3).

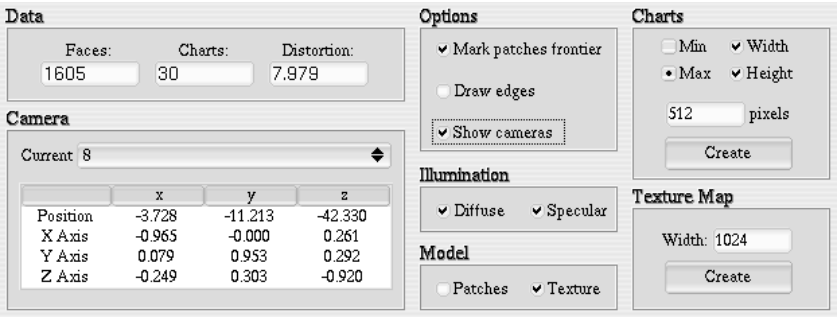


Figure 4.3: The toolbar with the atlas information and controls related to cameras (left), visualization options (center) and charts resolution and texture map construction controls (right).

4.1.2 The Charts View

As the model view, the charts view can be separated in two parts: a square OpenGL window which holds the chart that is being edited and a toolbar with many brush options (these options will be detailed in the next section).

After constructing the charts, based on the resolution given by the user, we construct the painting area of each chart: the charts window. When a user selects a chart to paint, the application places this chart centered in this window (for details, see section 4.1.4).

This window works as a 2D painting application. The user can select the brush from the toolbar and paint strokes (a stroke begins when the mouse button is pressed inside a chart boundary and continues until the pressure is released or the chart boundary is reached) as if he was painting using a 2D painting/editing software (like Gimp), which is very easy. The texture image associated to each chart is mapped onto a 2D OpenGL square with vertices at the OpenGL window corners. Figure 4.4 shows the charts window, with a stroke being painted.

The user paints strokes directly on the texture image associated to each chart. When a stroke is being painted and the mouse reaches a chart boundary, the application (if a user flag is set) automatically loads the neighbor chart in the window and moves the mouse to a position such that the stroke that is being painted could be continuous at the frontier between adjacent patches. This process will be detailed in Section 4.1.5.

4.1.3 The Brush Object

The brush is the main object of a painting application, since it contains the color information that will be added to the model, and how (if we are painting a solid color, a pattern, a color

Figure 4.4: The charts window of the attribute editing application, with a stroke painted on a chart (note the chart boundary in black).

with alpha, etc). For this purpose, we use a mouse to move a cursor around the screen. The hot spot of the cursor specifies the position of the brush.

Our brush object has the following flags and parameters:

- **Mask:** The mask is a matrix of weights for the brush color. It is modified when the user changes the shape, size, hardness and/or alpha of our brush object.
- **Color:** The brush color.
- **Erase:** A flag indicating if the brush is a rubber.
- **Shape:** The brush shape.
- **Pattern:** The image pattern of the brush.
- **Size:** The brush size (width and height).
- **Hardness:** The hardness which basically changes the function of the mask.
- **Alpha:** A constant weight that multipliers the mask.
- **Spacing:** The interval between the painting of two strokes.

The brush toolbar is showed in Figure 4.5. From that the user can choose the brush shape, pattern, options of the mask, color, and see the resulting brush on a viewport.

The value of each cell in the brush mask is the weight of the brush color. So, for example, if the user paints a one pixel brush that has a mask with the value w_b in its unique cell ($w_b < 1.0$) in the position (i, j) of the texture map, which has a color value $c_{old}(i, j)$, the new color in that position will be $c_{new}(i, j) = w_b \cdot brush.color + (1 - w_b) \cdot c_{old}(i, j)$.

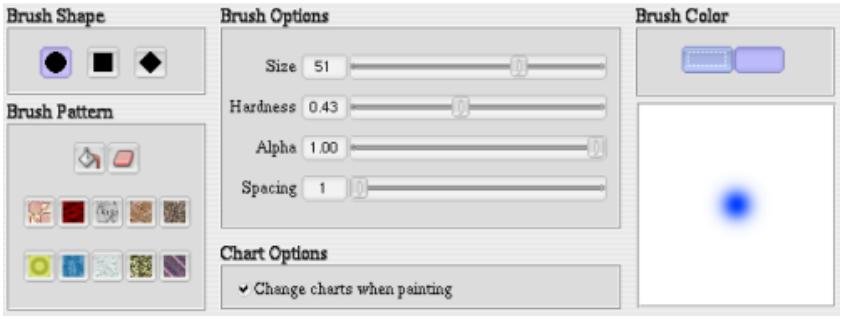


Figure 4.5: The brush toolbar.

Brush Shapes

We have implemented three different brush shapes: circle, square and diamond

Brush Patterns

Normally, in a classical 2D painting applications, the paint on the brush is a constant color. A good improvement that can be done is to allow the paint to vary as a function of position in a texture or pattern. We provide the patterns showed in Figure 4.6 for our application.

Figure 4.6: Patterns available in our application.

Patterns are indexed by the coordinates of the canvas (charts window). For instance, when the user begins painting a stroke using a pattern with dimensions (w, h) on the canvas, at position (x, y), the application repeats this pattern image over the canvas based on this position. If the position $(x + w, y)$ is reached, the application picks the color of the pattern as if the user was painting at position (x, y).

Brush Options

There are options to control the brush mask (size, hardness and alpha), to control the spacing between strokes (spacing) and to set the type of the painting:

- **Size:** The brush size (width and height). As we want to put the hot spot of the cursor int the center of the brush square area, we only allow odd values values for this option.

- **Hardness:** The hardness, when choose different from one, is basically the sigma parameter of a gaussian kernel centered at the brush mask center. This value is between 0 and 1.
- **Alpha:** A constant weight between 0 and 1 that multipliers the mask (i.e., the alpha channel for the brush mask).
- **Spacing:** The interval (in pixels) between consecutive brush marks when the user trace out a stroke.
- **Fill:** If selected, the application fills the chart area with the brush color or pattern, when the user clicks on the chart.
- **Erase:** If selected, the brush works as a rubber.

Given this options, we have a way to construct the mask equation. The mask value $m(x, y)$ of a brush with radius r, center (c_x, c_y), alpha a and hardness h is:

$$m(x,y) = a \cdot \exp\left(\frac{-((x-c_x)^2 + (y-c_y)^2)}{2 \cdot h^2 \cdot r^2}\right) \tag{4.1}$$

Figure 4.7 shows some brushes generated with different values for brush shape, size, pattern, size, hardness and alpha, and their respective masks painted on a cubic surface.

4.1.4 Creating a Texture Map

In this section we will detail the process of opening and discretizing a parametrized atlas, based on the atlas resolution given by the user. There are two kinds of atlases supported by our application:

1. An atlas created by the methods described in the Chapter 3, that is already discretized in a texture map. As we have detailed, this atlas is parametrized using the projective mapping of the REAL cameras associated to the charts.
2. An atlas created by the methods that will be detailed in Section 4.2. This atlas is parametrized using the projective mapping of the VIRTUAL cameras associated to the charts, but it was not discretized in a texture map.

For the first kind, since the atlas resolution comes from the input images, we do not have to re-scale the texture map. (we could re-scale this texture atlas in order to paint strokes with a resolution different from the input images, but we have not yet implemented this option). The texture map already has been created.

The second kind of atlas, differently to the first one, was not discretized in a texture map. We will do this job based on the parameters of the camera associated to each chart and the resolution given by the user.

As we will see in Section 4.2.2, for each patch we apply a transformation in order to translate the view position of its associated camera to the origin and aligns the camera axis with the canonical base through a rotation. This transformation, when applied to each

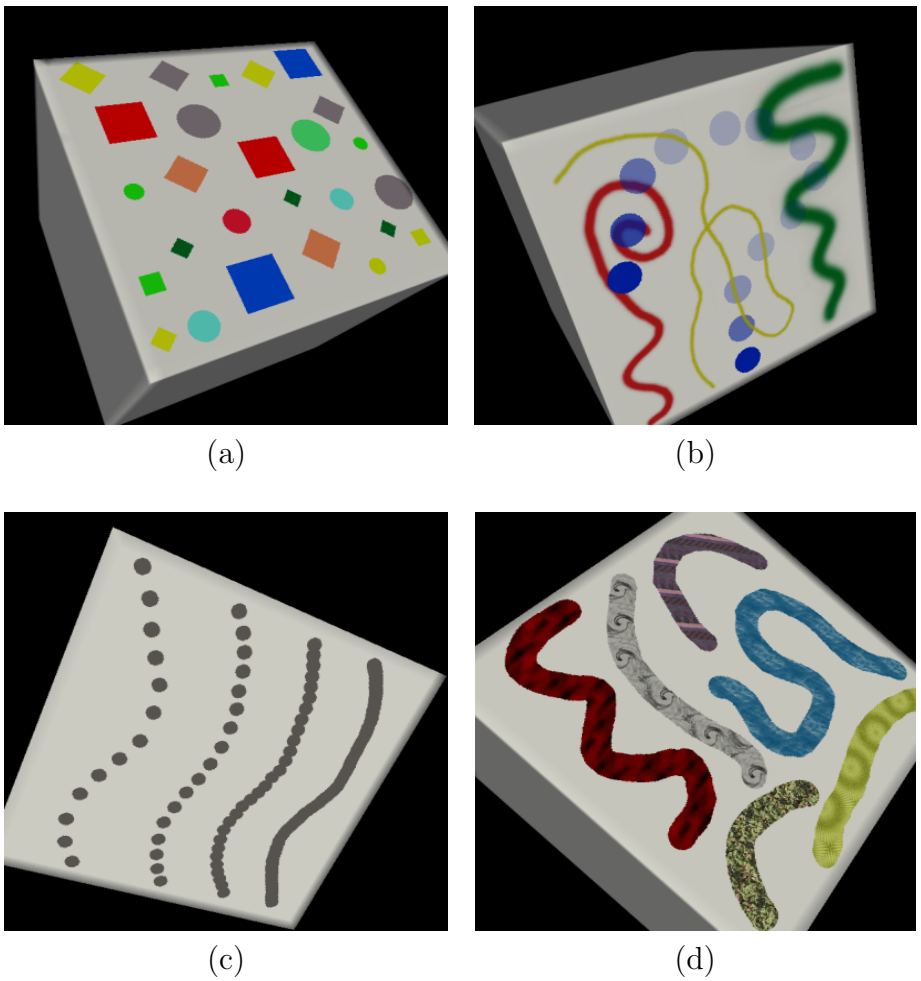


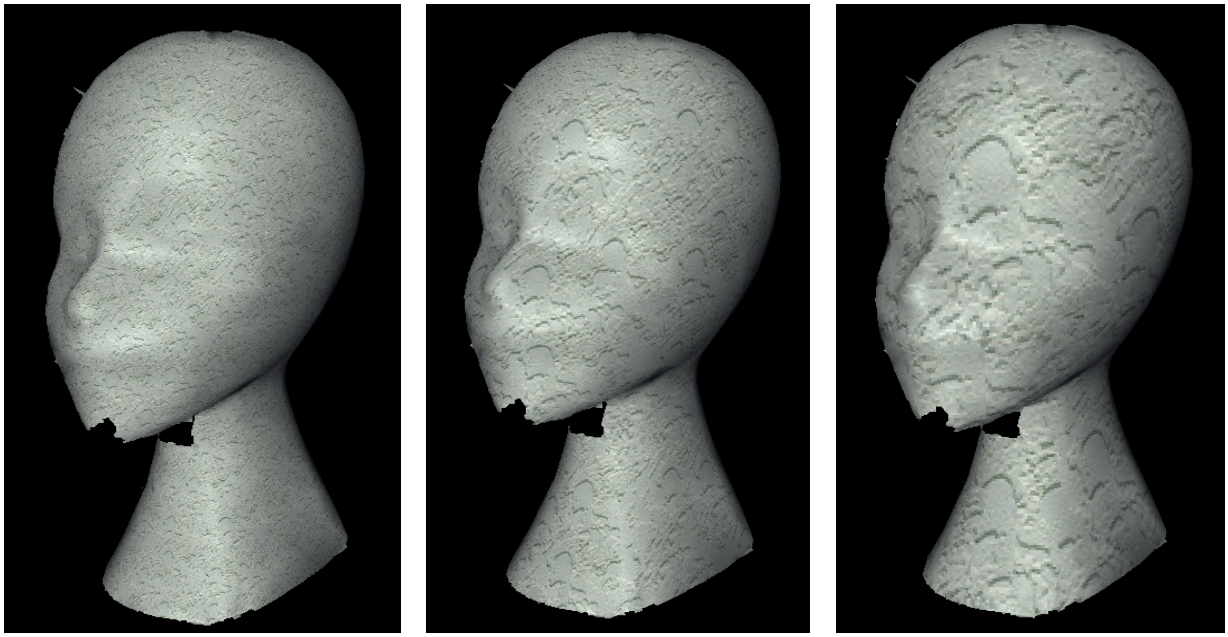
Figure 4.7: Strokes painted on a cube with different shapes, size and colors (a), different values for hardness and alpha (b), different values for the spacing between brushes (c) and different patterns (d).

patch, will give to us the patch 3D bounding box (minimum and maximal coordinates of its faces in x, y and z directions), i.e, a rectangular prism shaped viewing volume, since we are working with orthogonal cameras. We then create a unique view volume defined by the smallest enclosing cube of the 3D bounding boxes of all patches. This unique view volume is necessary because we have to maintain the scale coherence between the charts.

The atlas scale is calculated using the parameter given by the user: the maximum width/height of the charts. We then create an square OpenGL window (the charts window) with dimensions equal to this parameter. Using the transformation of the camera associated to each chart, and the orthogonal perspective transformation from the dimensions of the smallest enclosing cube calculated in the previous paragraph, we project each patch onto the square OpenGL window (obtaining in this way a discretized chart) and construct a square image associated to this chart (which we will call *Texture Image*). This image is the parametrization domain for each surface patch (as the input images of Chapter 3). When the user has finished his painting, he selects a texture map width and generate a texture map

for the atlas, packing the charts using the method described in Section 3.5.

Figure 4.8 shows three models textured with atlases of different resolutions.



(a) (b) (c)

Figure 4.8: The results of painting a pattern on a 3D model. The maximum width of the charts, given by the user, is 512 (a), 256 (b) and 128 pixels (c).

4.1.5 Painting Strokes

In the previous section we explained how we create a texture map based on a collection of parametrized charts. In this section we will detail how the strokes are painted on the charts. Since the user paints directly on the texture map, we have to develop a method such that a stroke painted on adjacent patches is continuous on the surface.

Beyond the texture map, we have other auxiliary images:

- **Texture Image:** The "front-buffer" of the painting application. For each chart we create a square texture image with dimensions given by the user. When the user paints a stroke on the OpenGL window this image is updated with this new stroke (only if the brush is inside the boundary of the chart).
- **Texture Image Input:** This image is useful if we want clear the strokes painted on a chart (a rubber tool). If there are input images associated to the charts, as in the previous chapter, this image the one associated to each chart. If there are not, this image is filled with the white color.
- **Texture Image Before:** This image is useful if we want to undo a painting. It keeps the Texture Image just before the user starts painting a stroke. When the mouse is released, we update this image.

- **Chart Adjacency Image:** Each chart is associated to a image in which the pixels define the boundary of the chart and the adjacent charts. In the region inside of the chart boundary the pixels values are the *id* of the chart. On the chart boundary the pixel values of this image are the *id* of its adjacent charts.

When a user selects a chart and a brush and press the mouse inside the chart boundary:

1. A stroke is calculated by combining the brush color (computed from its mask and options) and the color of the Texture Image.

2. The pixels of the Texture Image, in the region of the brush inside the chart boundary, are updated with this stroke.

3. This Texture Image is mapped onto a 2D OpenGL square with vertices at the OpenGL window corners (see Section 4.1.2).

4. The system re-projects the painted strokes in the model from the texture image of each chart according to the predefined texture coordinate of each vertex, and instantly presents the result in the 3D view.

In order to strokes painted on adjacent patches be continuous on the surface, the following method was developed. Suppose that a user begins to paint a stroke on a chart. The application automatically repeats the steps described just above. One issue is when to terminate a stroke. A stroke should always end when the brush is outside the charts boundary. Suppose that the user starts painting a stroke on a chart Θ. When the brush stroke center (the hot spot of the cursor) reaches a pixel p on the chart boundary, the application checks a user flag: automatically switch the charts when painting. If this flag is not set, nothing is done (remembering that we only paint strokes inside the chart boundary). If the user set this flag, then we have to update the charts window with a new chart. We decide which chart Θ_a will be placed in this window by analyzing the Chart Adjacency Image of the current chart in pixel p. The value of this pixel, as mentioned above, keeps the *id* of the adjacent chart in that boundary region.

Now that we know which chart will be loaded in the charts window, we have place the mouse hot spot on the pixel p_a of Θ_a corresponding to the pixel p of Θ. This pixel p_a is obtained in this way:

1. We compute the 3D point corresponding to p through the parameters of the camera associated to Θ (Algorithm 19).

2. We project this 3D point using the parameters of the camera associated to Θ_a in order to find p_a.

3. Place the brush stroke center in this p_a.

Figure 4.9 exemplifies this process.

Algorithm 19 find3DPointOnBoundary(Θ_a, Θ, p)

$E \leftarrow$ set of 3D edges of the boundary between the patches of Θ_a and Θ
$l \leftarrow$ 3D line obtained by computing the inverse projection of p using $\Theta.camera$
$P \leftarrow$ set of potential 3D points
for each edge e in E **do**
  $p \leftarrow$ 3D point of e that gives the minimum distance between e and l
  $P.insert(p)$
end for
$p_c \leftarrow$ the closest point to l from P
return p_c

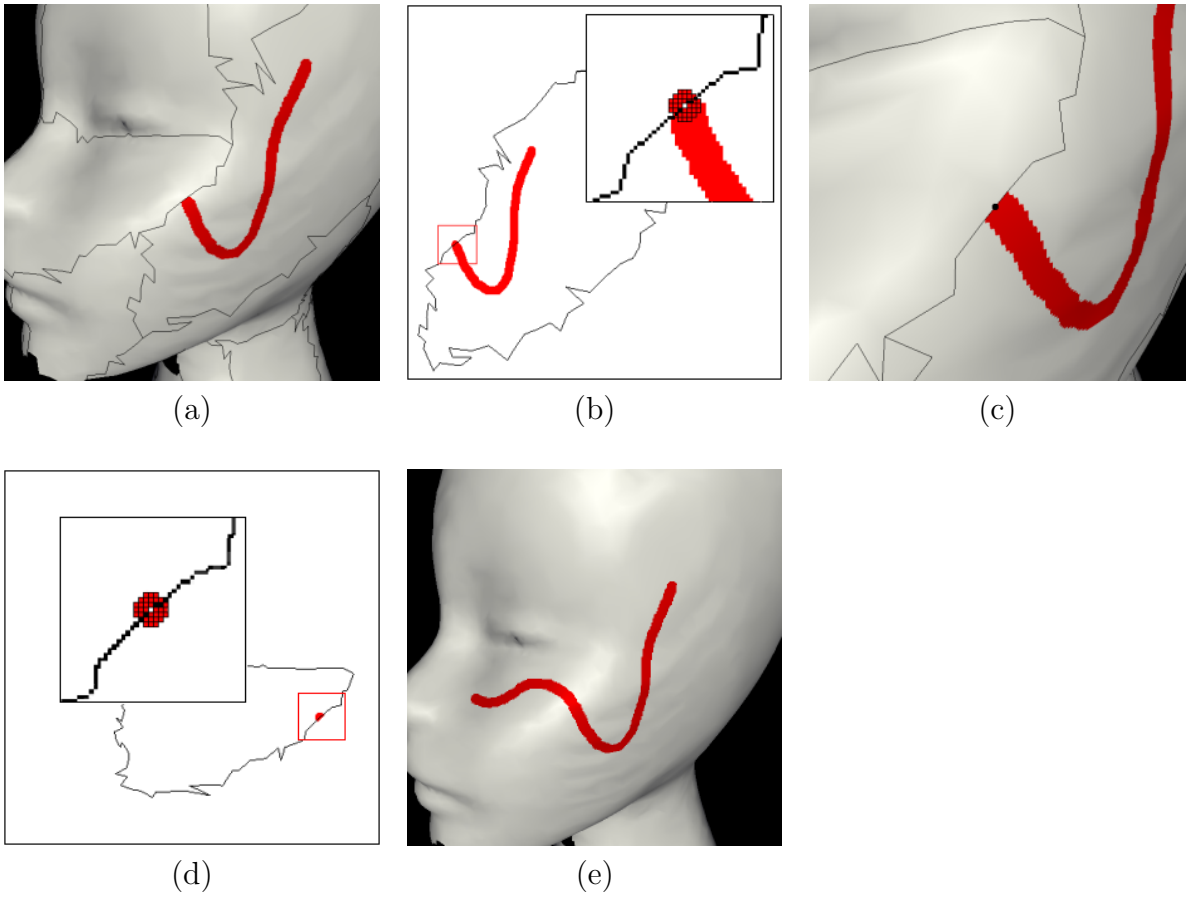


Figure 4.9: A user is painting a stroke on a 3D model (a) and the brush stroke center reaches a pixel on the chart boundary (b). From the center pixel of the stroke we compute the 3D point corresponding to this point through the parameters of the camera associated to the patch (c). We project this 3D point using the parameters of the camera associated to the adjacent patch and place the brush stroke centered at this position on the adjacent chart (d). In this way the stroke, which is being painted on different patches, is continuous on the surface (e).

4.2 Atlas Construction for Attribute Editing

In the previous section we detailed an attribute editing application based on a projective texture atlas. As mentioned before, a good projective texture atlas is a one that minimizes the distortion of the underlying surface-to-texture mapping.

A way to construct such atlas is to use the optimization methods for atlas construction described in Chapter 2. As we are dealing with projective mapping, the $\mathcal{L}^{2,1}$ metric is totally suitable for our problem, since it captures the shape and normal field of a surface.

We develop an algorithm that produces an optimal atlas and a set of virtual cameras based on a triangle mesh and some user-defined parameters. This output serves as the input for the attribute editing application.

4.2.1 Mesh Partitioning

In the way to make our atlas generation process interactive, we adopt an user and error-driven method. User driven because we let the user defines the minimum number of charts and/or maximum mapping distortion when constructing the atlas. Error driven because, given the error metric and a desired number of charts, we want to find a surface partition that minimizes the total distortion of the mapping.

The algorithm is very related to the algorithm described in Section 3.3, but with some differences. The first difference is that we do not have to worry about visibility when executing the patch growing step, since we will define the view position (and consequently the projective mapping) of each chart at the end of the process. The second difference is that in the method of Section 3.3 we construct a texture atlas from a set of images, what means that the parametrization of each chart comes from the projective mapping of its FIXED POSITION associated camera (real camera). Now, since we do not have a set of cameras and images (i.e. a previously defined projective mapping) as input, we could define, for each chart, its own parametrization.

The optimal atlas generation works as follows:

1. The user defines the minimum number of charts and/or maximum mapping distortion.

2. The phases *patchSeeding*() and *patchGrowing*() are repeated alternately until convergence, just as described by Cohen-Steiner *et al.* [9] (Section 2.1.3).

3. If the minimum number of patches and/or the maximum distortion has not been reached, add a new seed (like Sander *et al.* [23]).

4. Repeat steps 2 and 3 until the minimum number of patches and/or the maximum acceptable distortion is reached.

5. Then we check if the boundary of each patch projects injectively into the plane defined by the patch normal and barycenter (this is done by simply checking, for each patch, the signal of inner product between the patch normal and all face's normals of the patch. If one of these products is negative, the patch projection onto its plane is not injective). This checking is necessary because we will parametrize the charts using the projective mapping of the virtual cameras created from each patch normal.

6. Repeat steps 2 and 5 until the boundary of each patch projects injectively into the plane defined by the patch normal and patch barycenter.

7. Merge the patches if this does not violate the injectivity criterion.

The algorithm is described in Algorithm 20:

Algorithm 20 optimalAtlasConstruction()

```
mnp ← minimum number of patches
np ← number of patches
np ← 0
md ← maximum total distortion
d ← total distortion
d ← MAXFLOAT
while np < mnp or d > md do
  np ← patchAdding()
  while d is increasing do
    patchSeeding()
    patchGrowing()
    update d
  end while
end while
while one of the patches boundary do not project injectively into the plane defined by the
patch do
  np ← patchAdding()
  while d is increasing do
    patchSeeding()
    patchGrowing()
    update d
  end while
end while
patchMerging()
```

In the next sections we will explain each part of the algorithm.

Patch Adding

In order to bootstrap the algorithm, we first select, we first assign a random face to be the first seed. Then we iterate over *patchSeeding*() and *patchGrowing*() until convergence. if the minimum number of seeds and/or maximum mapping distortion has not been reached, we add a new seed, based on the following criteria, proposed by Cohen-Steiner *et al.* [9]:

1. For each patch select the one with maximum total distortion (based on the $\mathcal{L}^{2,1}$ metric).
2. For each face of this selected patch pick the face with worst distortion error as the initial seed for the next flooding.

This tends to place the new seed far away from the other seeds and in a region with maximum distortion.

When we add a new seed to the partition we clean up the list of faces of each patch (except the seeds) and, for each patch, make the patch normal and barycenter be its seed normal and barycenter.

Patch Seeding

The problem can be stated as follows:

Given a surface partition into patches, update the seed for each patch.

For each patch of the previous partition we want to select as the new seed for the next growing process the most similar face. This similarity is calculating using the $\mathcal{L}^{2,1}$ metric, described in Equation 2.3. With this metric we are able to distributes the patches according to local surface complexity. The distortion metric between a face f, with normal n and area s, and a patch P, with normal N_P, is:

$$E(f, P) = \|n - N_P\|^2 \cdot s \tag{4.2}$$

We visit each patch P and go through all its faces to find the one with the smallest distortion error $E(f, P)$. Then we update this face to be the new seed of that patch and clean up the list of faces of each patch (except the seeds).

Patch Growing

The problem can be stated as follows:

Given n seeds representing the n patches, assign each face to a patch.

We partition our mesh using the same flooding algorithm described in Section 3.3.1 (Algorithm 5). However, now we use a metric based only on the normal information of the faces and the patches, described in Equation 4.2.

Once each face has been assigned to a patch we wish to update, for each patch, its normal and barycenter, and the total distortion of the partition. For the $\mathcal{L}^{2,1}$ metric, the patch normal N_P is the area-weighted average of the face's normals (n is the normal of a face f and s is its area):

$$N_P = \sum_{f_i \in P} s_i \cdot n_i \tag{4.3}$$

The patch barycenter X_P is chosen to be the barycenter of the region defined by its faces. So, if x if the barycenter of a face f:

$$X_P = \frac{\sum_{f_i \in P} s_i \cdot x_i}{\sum_{f_i \in P} s_i} \tag{4.4}$$

With each patch normal updated, the total distortion of the partition is calculated, using the $\mathcal{L}^{2,1}$ metric in Equation 2.4. The phases *patchSeeding*() and *patchGrowing*() are repeated alternately until convergence.

Patch Merging

Since we do not aggregate in the metric used by the *patchSeeding*() and *patchGrowing*() phases the injectivity test (the test, as showed in Algorithm 20, is done after the convergence of the distortion), the process of adding charts based on this criterion may add an excessive number of charts in the atlas.

For this reason we can merge adjacent patches until we reach some of the user-defined parameters (minimum number of charts or maximum mapping distortion), if this operation does not violate the injectivity criterion. This merging method is inspired by Marinov and Kobbelt [21]. Being X, N and S the barycenter, normal and area of a patch P, respectively, a patch merge operation of two patches P_1 and P_2 computes a new patch P_m with:

$$N_m = \frac{S_1 \cdot N_1 + S_2 \cdot N_2}{\|S_1 \cdot N_1 + S_2 \cdot N_2\|},\ X_m = \frac{S_1 \cdot X_1 + S_2 \cdot X_2}{\|S_1 + S_2\|} \tag{4.5}$$

Basically the algorithm in each step performs the patch merge operation with the highest priority (the lowest error) that does not violate the injectivity criterion, and stops when the minimum number of charts or maximum mapping distortion is reached. The $\mathcal{L}^{2,1}$, in this case, is estimated by computing an area weighted sum of the deviation between the normals before and after a merge operation:

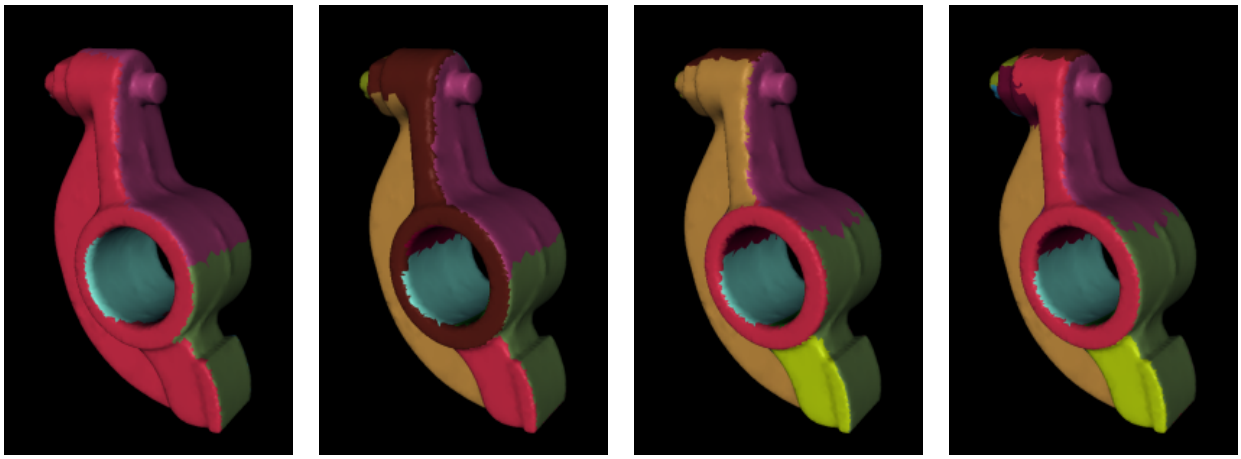
$$E(P_1, P_2, P_m) = S_1 \cdot \|N_1 - N_m\|^2 + S_2 \cdot \|N_2 - N_m\|^2 \tag{4.6}$$

The algorithm is exposed in Algorithm 21.

Algorithm 21 optimalPatchMerging()

```
Q ← priority queue
P ← list of patches
mnp ← minimum number of patches
np ← number of patches
md ← minimum distortion
d ← total distortion
for each patch pair of adjacent patches P_k and P_l in P do
  Q.push(P_k, P_l, E(P_k, P_l, P_m)) {Equation 4.6}
end for
while Q ≠ ∅ and np ≥ mnp and d <=> md do
  (P_k, P_l) ← Q.pop()
  if merge of P_k and P_l does not violate the injectivity criterion then
    patchMerge(P_k, P_l) {Algorithm 6}
    update patch neighbors
    update priorities
    update d
    decrease np
  end if
end while
```

In Figure 4.10 we show an example of the mesh partitioning algorithm on a model.



5 patches 10 patches 15 patches 20 patches

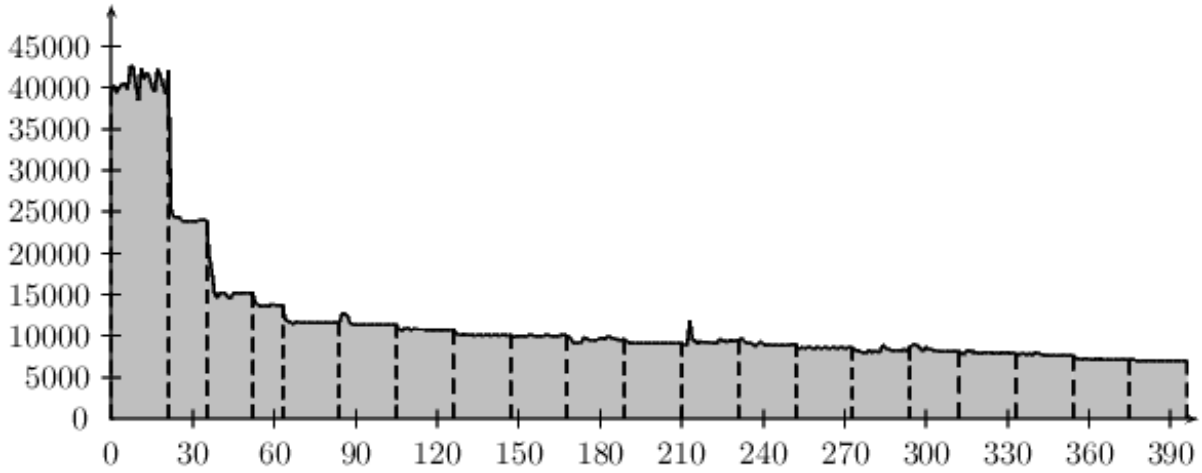


Figure 4.10: Applying the mesh partitioning algorithm on a model with 20k faces. On the top the process of adding patches to the partition . On the bottom a curve indicating the $\mathcal{L}^{2,1}$ distortion error as a function of the number of iterations. As expected, a few iterations, for each partition size, suffice to converge the distortion error

4.2.2 Atlas Parametrization

Once we have found a optimal partition of a mesh from a set of user-defined parameters, we have to find a way to parametrize this atlas. Since we are constructing an atlas to be used by our attribute editing application, an easy and natural solution is to parametrize it through projective mapping.

Therefore, for each patch of the mesh partition we associate a virtual camera. Each virtual camera has the following parameters:

- View Position
- View Direction

- Up Vector

From these parameters calculated (using the patch normal and barycenter, as we will explain in Algorithm 22), we obtain a camera transformation matrix, which translates the view position to the origin and aligns the camera axis with the canonical base through a rotation.

We have to choose now the projective transformation: perspective or orthogonal. Orthogonal projection is a better choice if the region we want to project does not have much distortion. Perspective projection is better if we want to simulate real cameras (lens distortion, focus, etc). Since our algorithm produces charts with small distortion, and we do not want to simulate real cameras, orthogonal projection is a better choice. This transformation, when applied to each patch, will give to us the patch 3D bounding box (minimum and maximal coordinates of its faces in x, y and z directions), i.e, a rectangular prism shaped viewing volume.

Since we want that the charts have the same scale, we have map the patches onto a 2D domain using the same viewing volume. This unique view volume is defined by a 3D bounding box with dimensions being the maximum and minimum coordinates among the 3D bounding boxes of all patches.

Algorithm 22 describes how to compute the virtual camera parameters for each patch. The scale of the charts, as we detailed, is chosen by the user in the attribute editing application (4.1.4).

Algorithm 22 computeOrthogonalCameras()

$P \leftarrow$ list of patches
for each patch p in P **do**
  $p.camera.viewPosition \leftarrow p.barycenter + p.normal$
  $p.camera.viewDirection \leftarrow -p.normal$
  $p.camera.upVector \leftarrow U - (U \cdot p.camera.viewDirection) \cdot p.camera.viewDirection$
  apply camera transformation on p, and obtain $p.bBox3D$
end for

The methods described here and in the previous section produce a optimal partition and a set of virtual cameras, that serve as the input for the attribute editing application detailed in Section 4.1.

Chapter 5

Results

In this chapter we demonstrate some results of our studies. In Section 5.1 we show the results obtained by processing 3D data acquired with a camera, following the methods described in Chapter 3. Section 5.2 reports the results of applying the atlas construction algorithm described in Section 4.2 and exposes the use of the attribute editing application to create texture maps.

5.1 Multiresolution Texture Maps from Multiple Views

We have applied the texture atlas construction from a set of images on two target objects, the *Branca* model, that was acquired from a set of 3 images, and the *Human Face* model (gently yielded by the ISTI-CNR), acquired from 6 images. These two models and respective input images are showed in Figure 5.2.

In Figure 5.3 we show the results of constructing texture maps for the *Branca* and the *Human Face* models. The results were obtained through the variational atlas construction pipeline detailed in Chapter 3, without applying the texture mapping compression algorithm described in Section 3.4. The hole process is really fast, and the results obtained were very good. For the *Branca* model we note that the color differences on the frontier zones between adjacent charts are evident, due to the extremely variance in its input images, and the mapping distortion is high, due to the small number of images for this model.

In Figure 5.4 we compare the results of the variational map construction pipeline with and without applying the texture mapping compression algorithm described in Section 3.4. Although the visual quality of the textured model, when the compression algorithm is not applied, is better, the reduction of the texture map occupied space is considerable.

Table 5.1 exposes general quantitative results of the atlas construction process on the two models. *Stretch efficiency* is the total surface area in 3D (sum of the patches area, calculated from the dot products between each triangle normal and the normal of its associated patch) divided by the total chart area in 2D. *Packing efficiency* is the sum of chart areas in 2D divided by the rectangular texture domain area. We can separate the packing efficiency into *intra-rectangle efficiency* (the sum of chart areas in 2D divided by the sum of rectangles areas) and *inter-rectangle efficiency* (the sum of rectangles areas divided by the rectangular texture domain area). Therefore *packing efficiency* = *intra-rectangle efficiency*

· *inter-rectangle efficiency*. Figure 5.1 shows an illustrative example of these measures. For these results, we have ignored the overhead that would be caused by the 1-texel gutter required between charts (see Section 3.5). The method's efficiency, called *texture efficiency* is the *stretch efficiency* times the *packing efficiency*. Since the atlas produced for the *Human Face* model has almost 8 times the number of charts of the one produced for the *Branca* model, the packing efficiency and texture efficiency of the *Human Face* model is smaller.

Figure 5.5 and 5.6 shows an accuracy comparison between the texture models obtained by our algorithm and Callieri *et al.* [7] with the original object images. Despite of the good accuracy of the two methods (except for the *Branca* model, for which Callieri *et al.* [7] method have presented worse results, probably originated in the blending phase), we pose texture atlas generation as an optimization one, in way that our method tries to generate a partition that minimizes the stretch distortion and the number of charts. For this reason our method produces less charts than Callieri *et al.* [7], as showed in the figures.

In Callieri *et al.* [7] the texture packing phase is done with an optimization: they aggregate patches mapped to the same images if this reduces the overall texture space (if the resulting bounding rectangle is smaller than the sum of the independent ones). We compare the number of charts generated by our method with the number of charts generated by Callieri *et al.* [7] without (the real partition of the object) and with this optimization. In both models and cases we produce a lesser number of charts.

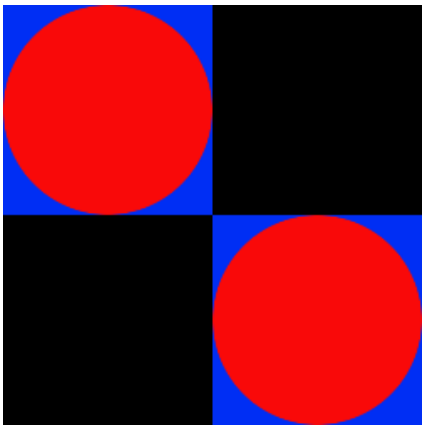
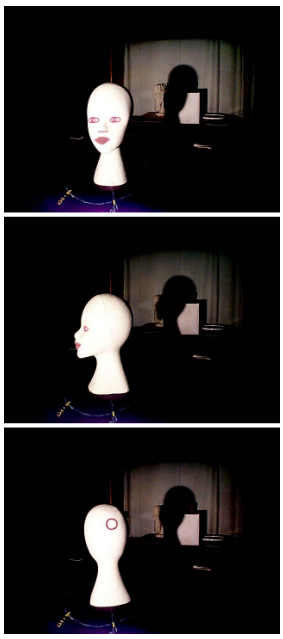
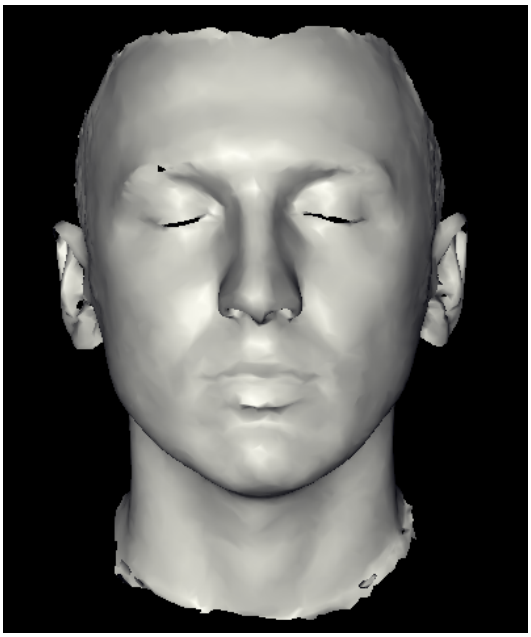


Figure 5.1: An illustrative example of how *packing efficiency* is calculated. The figure represents the texture domain, where the red circles represent the charts and the blue squares the bounding boxes. From the figure we conclude that the *intra-rectangle efficiency* is $\pi/4$ =78% and the *inter-rectangle efficiency* is 50%.

9852 triangles



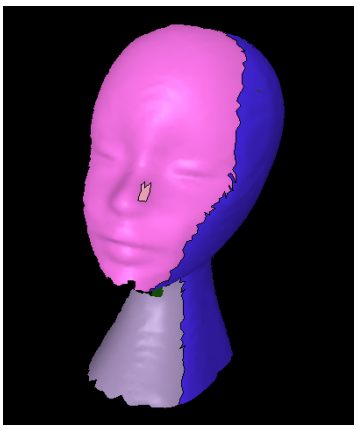
3 images



9406 triangles

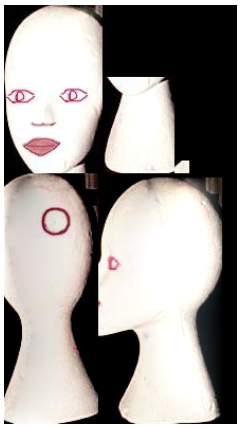
6 images

Figure 5.2: The two models and their set of images (on the top row the *Branca* model and on the bottom row the *Human Face* model.

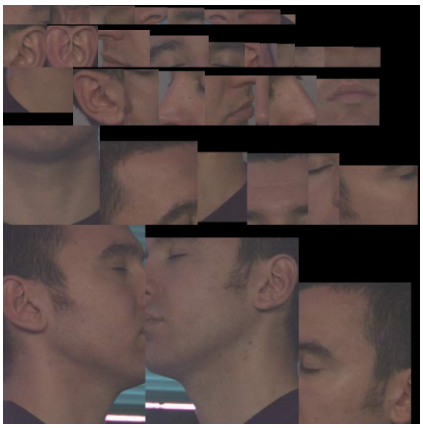


(5 patches, distortion = 5875.18)

(39 patches, distortion = 4680.54)



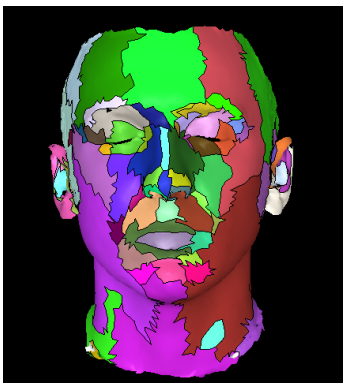
(dimensions=220×396)



(dimensions=750×755)

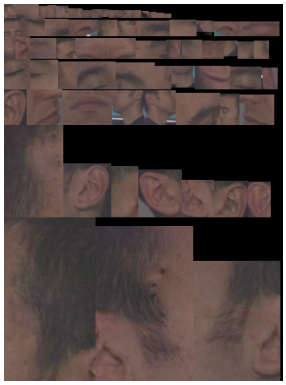
Figure 5.3: Applying the variational map construction pipeline on the *Branca* model (left) and on the *Human Face* model (right).

(39 patches)



(70 patches)

(dimensions=750×755)



(dimensions=320×433)

Figure 5.4: Applying the variational map construction pipeline without (on the left) and with (on the right) the texture mapping compression algorithm, on the *Human Face* model. The higher frequency regions (eyes, hair, mouth, etc) are preserved. In this case the reduction of the texture map occupied space is about 75%.

Models	Branca	Human Face
Vertices	5177	4971
Faces	9852	9406
Charts	5	39
Distortion	5875.18	4680.54
Texture map dimensions	220×396	750×755
Stretch efficiency	80%	86%
intra-rectangle efficiency	64%	49%
inter-rectangle efficiency	78%	80%
Packing efficiency	50%	39%
Texture efficiency	40%	34%

Table 5.1: Quantitative results

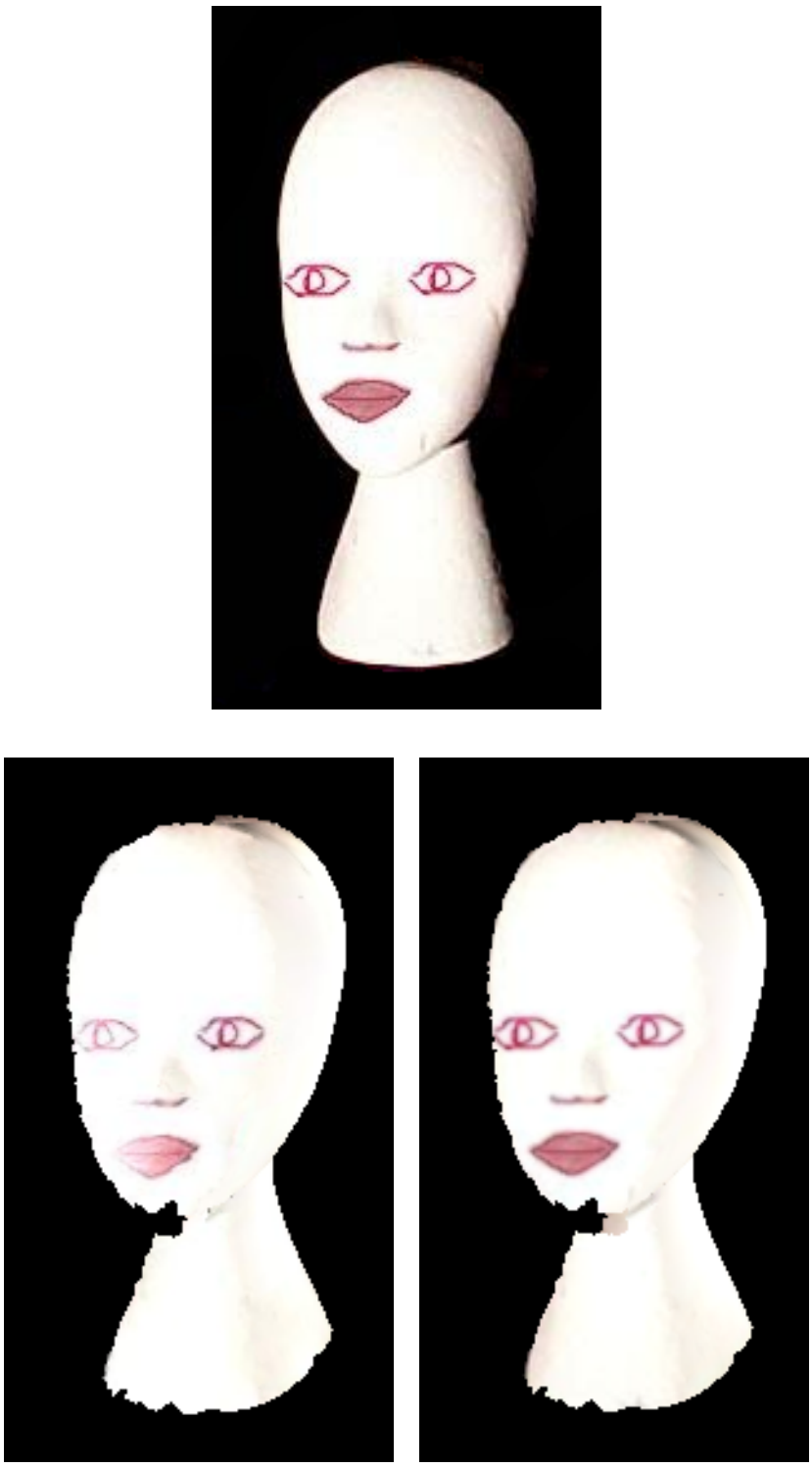


Figure 5.5: Photograph of the *Branca* object (a), the synthetic model by Callieri *et al.* [7] (256×512 texture map, 5 charts with optimization, 6 without) (b) and our synthetic model (220×396 texture map, 5 charts) (c).

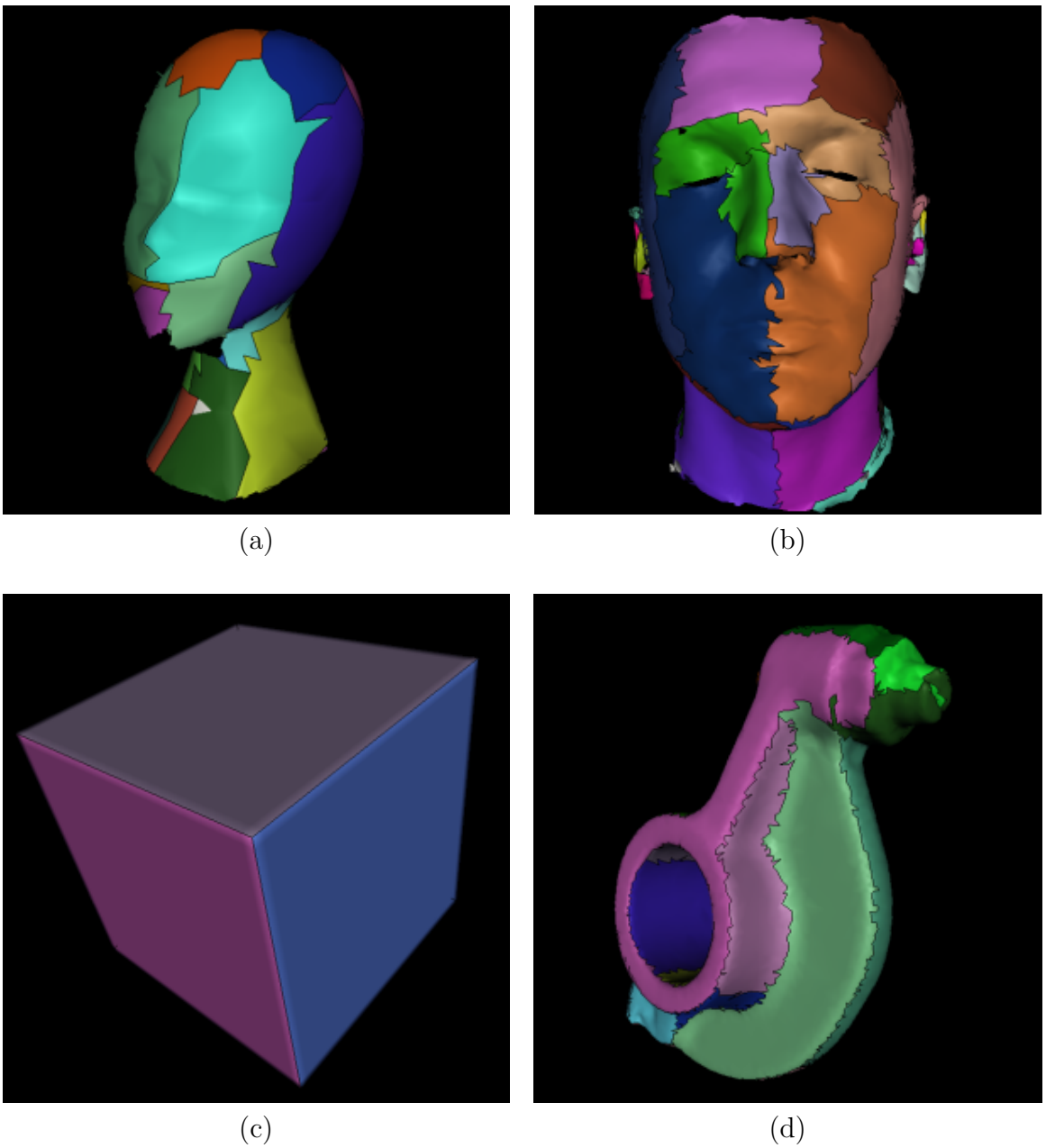
Figure 5.6: Photograph of the *Human Face* object (a), the synthetic model by Callieri *et al.* [7] (512×1024 texture map, 52 charts with optimization, 73 without) (b) and our synthetic model (750×755 texture map, 39 charts) (c).

5.2 Construction of Texture Maps Using the Attribute Editing Application

In this section we report some results obtained by the methods exposed in Chapter 4. First we show some results when applying the atlas construction algorithm for attribute editing (4.2). After that we demonstrate the use of the attribute editing application to create texture maps.

Figure 5.7 shows the results of the atlas construction for attribute editing on four models: *Branca*, *Human Face*, *Cube* and *Rocker Arm*. By looking to the *Branca* and *Human Face* models we note that the $\mathcal{L}^{2,1}$ distortion metric captures the symmetry of the models, and by looking to the *Cube* and *Rocker Arm* models we see that the local planarity of the models was captured. Table 5.2 exposes general quantitative results of this algorithm on the four models. An observation has to be done. Although it would be expected to the *Rocker Arm* model stretch efficiency be higher than in the *Branca* and *Human Face* models (due to its piecewise planar nature), we would have that to add more patches to achieve a higher stretch efficiency. Other observation on this model is its low packing efficiency, consequence of its non-convex form and presence of holes.

Using the atlases created by the variational atlas construction algorithm we show some texture maps constructed through the attribute editing application. In Figure 5.8 we show the process of painting on a *Screwdriver* model. In Figure 5.9 we show how our application can be used to paint details in the internal parts of models with holes, what would be very difficult if we were using a application that paints directly on the surface. Figure 5.10 shows an example of how the variational atlas construction algorithm minimizes the distortion of the underlying surface-to-texture mapping (i.e., texture stretch).



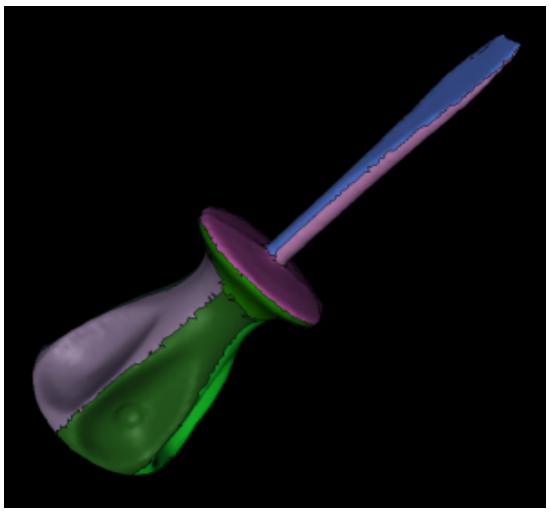
(a) (b)

(c) (d)

Figure 5.7: Applying the atlas construction on four models, with a user-defined number of charts of 30 (except for (c), which has 6 charts).

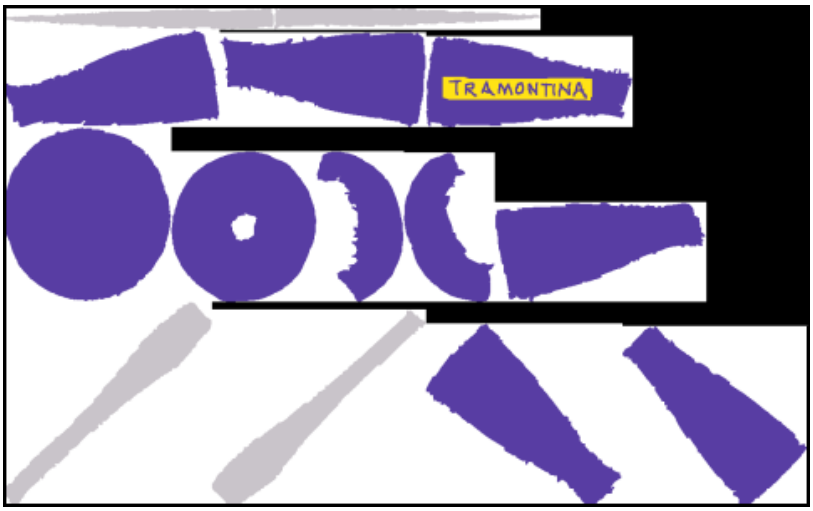
Models	Branca	Human Face	Cube	Rocker Arm
Vertices	1243	4971	5402	10044
Faces	1605	9406	10800	20088
Charts	30	30	6	30
Distortion	792.15	1896.17	0	6962.05
Texture map dimensions	512×821	512×939	512×1280	512×557
Stretch efficiency	93%	86%	100%	89%
intra-rectangle efficiency	51%	50%	100%	35%
inter-rectangle efficiency	76%	82%	100%	62%
Packing efficiency	39%	41%	100%	22%
Texture efficiency	36%	38%	100%	20%

Table 5.2: Quantitative results



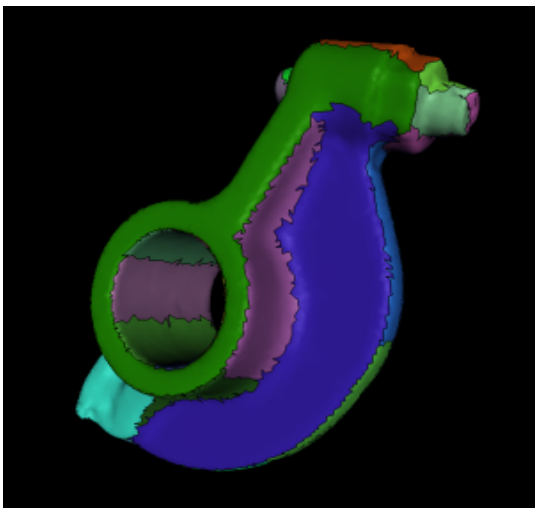
Mesh partition, 14 charts
(distortion=6159.48; stretch efficiency=90%)

Textured model

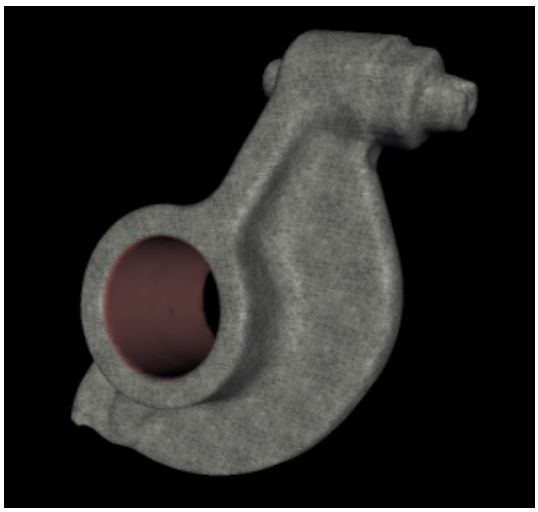


Texture map
(dimensions=1270×790; packing efficiency=37%)

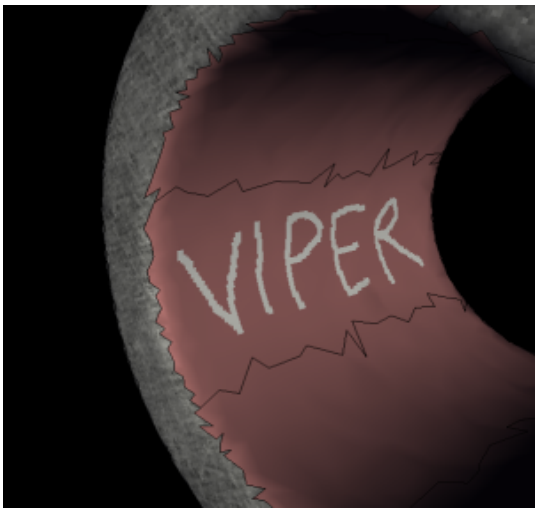
Figure 5.8: Painting on a *Screwdriver* model.



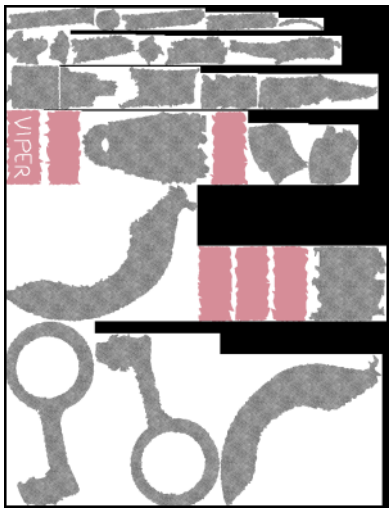
(a) 30 charts
(distortion=6962.05;
stretch efficiency=89%)



(b)

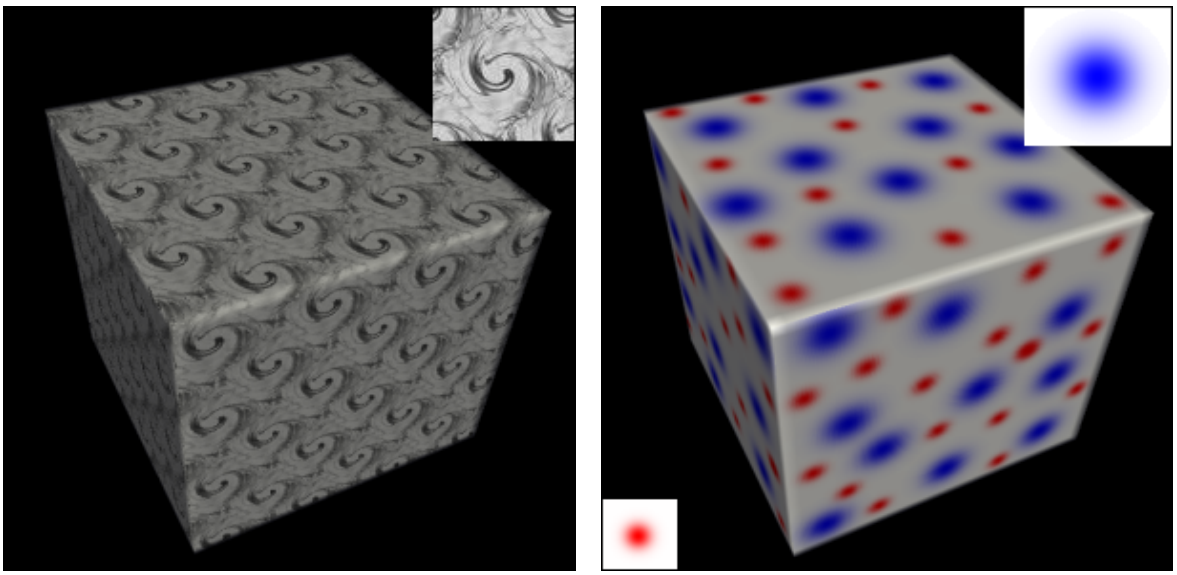


(c)



(d) Texture map
(dimensions=512×557;
packing efficiency=22%)

Figure 5.9: Painting on the *Rocker Arm* model. We apply the atlas construction algorithm for 30 charts (a) and paint this mechanical part with a metallic pattern (b). Since the painting is done directly on the charts, and our distortion metric minimizes the texture stretch, it is very easy to add details (such as mechanical specifications and brand) in the internal part of the model (c).



(a) (b)

Figure 5.10: Using the attribute editing application to paint a pattern (a) and circular strokes (b) in the atlas constructed for the *Cube* model. Since the $\mathcal{L}^{2,1}$ distortion for this atlas, as it was seen in Table 5.2, is 0, our application allows the user to construct a texture map with no stretch (the texture pattern is mapped in the same pattern on the surface and circular strokes are mapped in circular strokes, not ellipsoidals, on the surface

Chapter 6

Conclusions and Future Works

6.1 Conclusions

In this work we have proposed the use of a variational optimization scheme in the construction of texture atlases for 3D photography and attribute editing. With this scheme we were able to construct optimal texture atlases, minimizing the number of charts, the atlas distortion and the texture map occupied space.

In Chapter 2 we have described the variational shape approximation method detailed in Cohen-Steiner *et al.* [9] and explained how to use this scheme to construct texture atlases with small distortion.

We have developed in Chapter 3 a method for generating multiresolution texture atlases from a set of images, inspired by the pipeline created by Callieri *et al.* [7]. We have used a variational method to construct well partitioned and low distorted texture atlases. In addition, we have applied texture compression and packing techniques, to reduce the occupied space of charts, and blending methods, to increase the color coherence between adjacent patches.

Finally, in Chapter 4, we have developed an attribute editing application that allows the user to create and manipulate texture atlases. We have created a friendly interface that allows the user to directly paint on the charts of the atlas, with different types of pigments and patterns, as if he was painting using a 2D painting/editing software. For this application we have presented a variational method for generating a texture atlas and a set of virtual cameras, based on the ideas of Chapter 2.

6.2 Future Work

There remain a number of areas for future work:

- We have implemented a simplified packing algorithm (see Section 3.5). We would have better results, with respect to packing efficiency, if we pack the charts boundary directly rather than their bounding rectangles (like Lévy *et al.* [17] and Sander *et al.* [23]).

- Examine how best to address the trade-off between texture atlas distortion and the number of charts, for the atlas construction for attribute editing algorithm (Section

4.2).

- Exploit the construction of atlases with other surface attributes, like normal and displacement fields, in order to allow the user to create/modify any kind of attribute maps through the attribute editing application.
- Modify the attribute editing application such that it could be used for modeling operations, based on normal maps created/modified by a user, as exemplified in Figure 6.1.

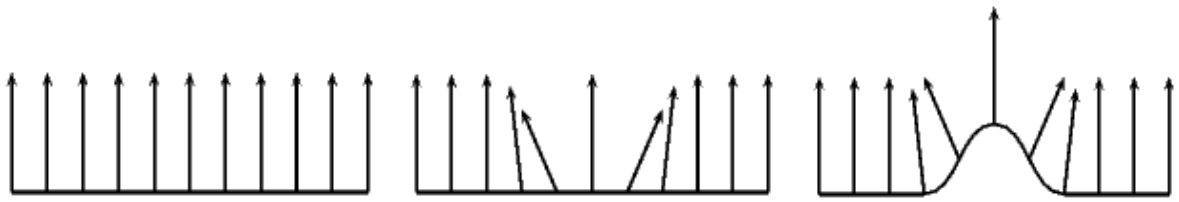


Figure 6.1: On the normal map of a planar surface (a). The attribute editing application could allow the user to modify the normal map (b) and, in addition, the geometry of the surface (c).

Bibliography

[1] Agarwal, Pankaj K., and Subhash Suri. Surface approximation and geometric partitions. In *SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 24–33, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[2] Balmelli, Laurent, Gabriel Taubin, and Fausto Bernardini. Space-optimized texture maps. *Comput. Graph. Forum*, 21(3):411–420, 2002.

[3] Bernardini, Fausto, Ioana M. Martin, and Holly Rushmeier. High-quality texture reconstruction from multiple scans. *IEEE Transactions on Visualization and Computer Graphics*, 7(4):318–332, 2001.

[4] Botsch, M., S. Steinberg, S. Bischoff, and L. Kobbelt. Openmesh - a generic and efficient polygon mesh data structure, 2002.

[5] Briggs, Willian L. A multigrid tutorial. `http://www.llnl.gov/CASC/people/henson/mgtut/welcome.html`.

[6] Burt, Peter J., and Edward H. Adelson. The laplacian pyramid as a compact image code. pages 671–679, 1987.

[7] Callieri, Marco, Paolo Cignoni, and Roberto Scopigno. Reconstructing textured meshes from multiple range rgb maps. In *VMV*, pages 419–426, 2002.

[8] Carr, Nathan A., and John C. Hart. Painting detail. *ACM Trans. Graph.*, 23(3):845–852, 2004.

[9] Cohen-Steiner, David, Pierre Alliez, and Mathieu Desbrun. Variational shape approximation. *ACM Trans. Graph.*, 23(3):905–914, 2004.

[10] Du, Qiang, Vance Faber, and Max Gunzburger. Centroidal voronoi tessellations: Applications and algorithms. *SIAM Rev.*, 41(4):637–676, 1999.

[11] Figueiredo, L. H., C. A. da Silva, and R. de B. Seixas. Interpolação de curvas de nível por difusão de calor. In *GeoInfo 2001*, pages 57–62, 2001.

[12] Gortler, Steven J., Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In *Proceedings SIGGRAPH '96*, pages 43–54. ACM Press, 1996.

[13] Hanrahan, Pat, and Paul Haeberli. Direct wysiwyg painting and texturing on 3d shapes. In *Proceedings SIGGRAPH '90*, pages 215–223. ACM Press, 1990.

[14] Hunter, Adam, and Jonathan D. Cohen. Uniform frequency images: adding geometry to images to produce space-efficient textures. In *Proceedings VIS '00*, pages 243–250. IEEE Computer Society Press, 2000.

[15] Johnston, Scott F. Lumo: illumination for cel animation. In *Proceedings NPAR '02*, pages 45–ff. ACM Press, 2002.

[16] Lévy, Bruno, and Jean-Laurent Mallet. Non-distorted texture mapping for sheared triangulated meshes. In *Proceedings SIGGRAPH '98*, pages 343–352. ACM Press, 1998.

[17] Lévy, Bruno, Sylvain Petitjean, Nicolas Ray, and Jêrome Maillot. Least squares conformal maps for automatic texture atlas generation. In *Proceedings SIGGRAPH '02*, pages 362–371. ACM Press, 2002.

[18] Lloyd, Stuart P. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–136, 1982.

[19] Maillot, Jêrome, Hussein Yahia, and Anne Verroust. Interactive texture mapping. In *Proceedings SIGGRAPH '93*, pages 27–34. ACM Press, 1993.

[20] Mallat, Stephane, and Sifen Zhong. Characterization of signals from multiscale edges. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(7):710–732, 1992.

[21] Marinov, Martin, and Leif Kobbelt. Automatic generation of structure preserving multiresolution models. *Comput. Graph. Forum*, 24(3):277–284, 2005.

[22] Milenkovic, Victor J. Rotational polygon containment and minimum enclosure. In *SCG '98: Proceedings of the fourteenth annual symposium on Computational geometry*, pages 1–8, New York, NY, USA, 1998. ACM Press.

[23] Sander, P. V., Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. In *Proceedings SGP '03*, pages 146–155. Eurographics Association, 2003.

[24] Sander, Pedro V., John Snyder, Steven J. Gortler, and Hugues Hoppe. Texture mapping progressive meshes. In *Proceedings SIGGRAPH '01*, pages 409–416. ACM Press, 2001.

[25] Velho, Luiz. A48. `http://w3.impa.br/~lvelho/a48/main.html`.