MASTER IN COMPUTER SCIENCE

Refining Multi-view 3D Reconstruction with Differentiable rendering

Master Thesis

Simone Raimondi

University of Bern

November 2017

UNIVERSITÄT BERN

UNIVERSITÉ DE NEUCHÂTEL

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Abstract

In this thesis we explore the possibilities of using a volumetric differentiable renderer for addressing the multi-view 3D reconstruction problem. Multi-view stereo has achieved great success after decades of effort. Recently, the reconstruction quality has improved considerably via shape-from-shading. Instead of taking the illumination and albedo information into account during the reconstruction process, we design a differentiable renderer that can directly minimize the reconstruction loss with respect to the interested rendering parameters such as geometry by back propagation. Our work is inspired from OpenDR[1], a mesh-based differentiable renderer, but uses a multi-scale Sign Distance Function (SDF) as the representation of the geometry that is rendered by a ray tracer written in C++. Experiment results show the superiority of our method in recovering complex geometric features even under large topological changes and with a few views of the objects.

Prof. Dr. Matthias Zwicker, Dept.of Computer Science, University of Maryland, Supervisor
Shihao Wu, Computer Graphics Group, University of Bern, Assistant

Contents

1 Introduction

Capturing digitalization of objects from the real world is a topic of great interest in today's world. Applications such as 3D printing also demand for high quality reconstruction to be able to reproduce realistic models. Multi-view stereo is one of the most popular 3D capturing approach where the input is a set of images of the target model and the output is a digital model. During the years, many methods based on multi-view stereo have been proposed [2] and they are able to achieve great results. The problem of these methods is that they are often very sensitive to noise, therefore they require some regularization term to be robust. This, however, tends to smear out the fine details in the reconstructed geometry.
To avoid this, methods based on shading variation have been developed. From one of the first proposed by Wu et al. [3], which was only working on objects of constant albedo, algorithms have evolved. Current state of the art can be found in the work of Langguth and al. [4], where the two approaches are combined to achieve better results. Our work expands from here since we did notice that by using the work of Langguth and al. [4], often the results are smoothed out and all the details are lost, especially with few views.
We were also inspired by the research of M. Loper and M. Black [1] where they introduce a framework for differentiable rendering. We present a new way of doing the 3D reconstruction using a multi-scale signed distance function as our geometric representation. The main focus of the project is to show that it is possible to capture fine details and topological changes in the SDF representation, when comparing the results of a rendering done with ray tracing of our geometry, to a set of target images. The SDF (signed distance function) representation is very adaptive to changes in the topology of the starting geometry compared to the target one. The challenges of using a SDF as representation are: *(1)* the rendering is not as simple as for triangle meshes, *(2)* how to compute the changes of our stored values to match our target geometry and *(3)* respect the SDF properties explained in more details later in the text.
The main contributions we provide are:

- a volumetric differentiable ray tracer able to render signed distance functions
- show the advantages of using this representation for different aspects (e.g. topological changes in geometry)
- demonstrate that our multi-scale approach is able to capture small details in the geometry even with few views of the object

We now introduce the complexity of multi-view 3D reconstruction, some theory on signed distance function and differentiable rendering.

1.1 Multi-view Stereo

Reconstructing 3D geometry from images is a classic challenge of Computer Vision and has occupied researchers for more than 30 years by now. The increasing quality of cameras allows to capture very high resolution images of the real world that can be used to generate high quality 3D contents.
The image-based 3D reconstruction task can be formulated as: *"Given a set of real or synthetic generated images, estimate the 3D geometry that best generates them, given that the camera positions, the material and lighting of the scene are known"*. This is shown in fig. 1.1.

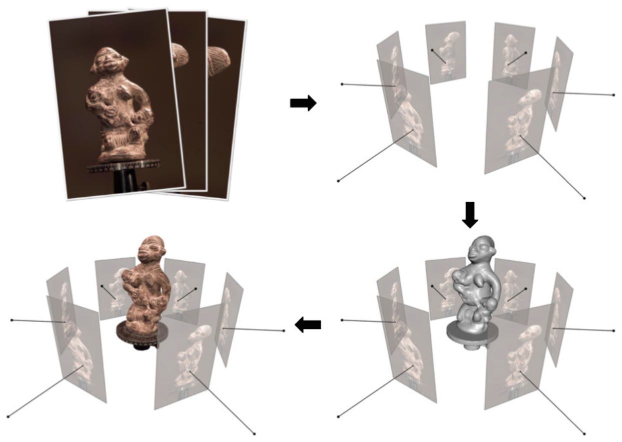

Figure 1.1: Multi-view stereo reconstruction setup. Clockwise: input images, positioned images, reconstructed 3D geometry and textured 3D geometry. Image from *Multi-View Stereo: A Tutorial* by Yasutaka Furukawa and Carlos Hernndez.

The definition by itself shows its complexity. Especially if the assumptions we make are not met, then the problem becomes ill-posed and multiple combinations of camera position, material, lighting and geometry can produce the same final image. This implies that there is no method to produce the correct 3D geometry from images alone. In fact many of the existing methods make some of these assumptions. Typically on the light model used for the scene, usually the Lambert model is used for the object. In fact, even state of the art methods as [4] still make the same assumptions. This comes from the fact that the Lambert model is very simple but is still able to reproduce the appearance of a lot of matte real objects. Including indirect illumination would make the problem much harder to solve.
The main idea behind multi-view stereo is that, given a set of images, if we know the camera parameters we can find correspondences points in the images that represent the same 3D point. This approach is called Structure-from-motion and is depicted in fig. 1.2. We can see how the same point $\mathbf{x_1}$ is projected to different coordinates in camera space for $\mathbf{Image\ 1}$ and $\mathbf{Image\ 2}$ and it's not visible in $\mathbf{Image\ 3}$. The coloured corner in the images shows the known correspondences between certain 2D points in the images. The difficult part in solving this problem is to find the actual correspondences between points in the images. This is usually done with some *features extraction* algorithm. Many algorithms have been developed during the years for this task, one of the most famous is SIFT [5].

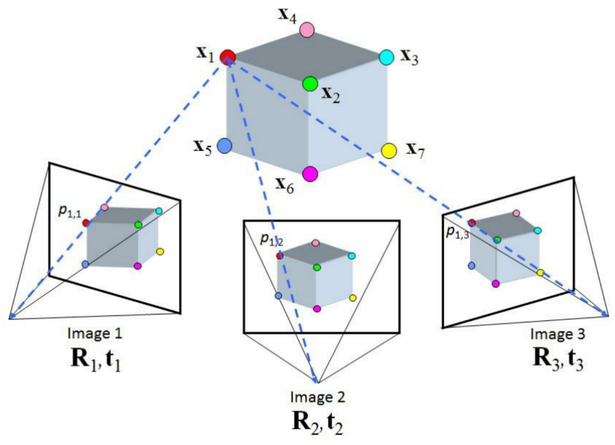

Figure 1.2: Structure-from-motion problem setup. Image from Ozgur et al. [6].

Nowadays, there are methods capable of reconstructing very large 3D objects from unstructured images, using cameras with different parameters. One of the most famous example is the work from S. Agarwal et al. [7].

1.2 Differentiable rendering

Usually a renderer is designed with the idea that, given some parameters Θ which can represent the scene geometry description, lighting, camera position and much more, we produce an image I. We can also express it in a more mathematical form as

$$f(\Theta) = I \tag{1.1}$$

where $f(\Theta)$ is our forward rendering process. $f(\Theta)$ can be many different functions, each one with a higher or lower level of realism in the produced final image.
When trying to reconstruct geometry from images, the target image (or more than one) we would like to have as final rendering is given. So we need to solve the optimization problem

$$E_I(\Theta) = |f(\Theta) - I|^2 . \tag{1.2}$$

To minimize this energy we need to get all the derivatives of $E(\Theta)$ with respect to the components of Θ. This is exactly what a *differentiable renderer* does. It is both able to produce pixel values given a description of our scene and provide the derivatives of each pixel, or the difference between the produced image and a target one, with respect to the input parameters Θ.
In our project, we rely on a simple implementation of the ray tracing algorithm for our rendering function f and we use *automatic differentiation* to compute the derivatives for backpropagation. We will describe our rendering process in more details later.

1.3 Signed distance function

As stated before we are the first to use a signed distance function to represent our geometry in a differentiable renderer. Here we introduce some of the basic concepts about it and will go into more detail during the method explanation when needed. A good reference on signed distance function, level-set methods and applications on them is [8].

A *distance function* is and implicit function defined as

$$d(\vec{x}) = min(|\vec{x} - \vec{x}_I|) \text{ for all } \vec{x}_I \in \partial\Omega \tag{1.3}$$

where $\partial\Omega$ is the *zero isosurface*. One thing we can already notice from this definition is the possibility that multiple points $\vec{x}_I$ exist for which the above condition is true. This has some important consequences on the computation of ∇d which might not exist at certain points.

A *signed distance function* is an implicit function ϕ where $|\phi(\vec{x})| = d(\vec{x})$ for all $\vec{x}$. Thus, $\phi(\vec{x}) = d(\vec{x}) = 0$ for all $\vec{x} \in \partial\Omega$, $\phi(\vec{x}) = -d(\vec{x})$ for all $\vec{x} \in \Omega^-$ and $\phi(\vec{x}) = d(\vec{x})$ for all $\vec{x} \in \Omega^+$. Ω^- is the internal part of the function, while Ω^+ is the outside.

Another important property of signed distance function is

$$|\nabla\phi| = 1 \tag{1.4}$$

which means that the length of the gradient at each point in our signed distance function must be equal to 1. Equation (1.4) is crucial, since it will play an important role later in our proposed method. To explain eq. (1.4) in an intuitive manner, it is clear that if we are on a certain isosurface at distance ϵ and we move along the normal, we expect to end on the next isosurface at distance $\epsilon + 1$. Again, it's important to keep in mind that eq. (1.4) is true only in a theoretical sense but fails at points that are equidistant from two or more points on the surface. In our case, since we are working on a Cartesian 3D grid, this problem will numerically smear out and we don't need to worry about having undefined derivatives. We will explain how we represent and store our SDF in more details later.

2 Related Work

Surface reconstruction has been a field of research for many decades now, and various approaches have been developed for different forms of input. Our technique uses a set of input images for which the cameras parameters are known. We will review here the many approaches and acknowledge their limits, true source of inspiration for us and our work and the proposed method.

2.1 Multi-view stereo

Multi-view stereo algorithm [2] is one of the most general passive reconstruction method. Approaches such as the one from S. Agarwal et al. [7] have shown that we can reconstruct geometry even from a very large and unstructured dataset from the Internet. Multi-view stereo approaches usually require a form of regularization to deal with areas where the structure is inconsistent and the stereo term of the energy minimization, based on photo consistency, is not able to match them. The recent work from Langguth et al [4] represents the state of the art by combining multi-view stereo and shape-from-shading methods. They propose a method based on the fact that stereo correspondences are more accurate in regions where the image gradient is large, while shape-from-shading works best in flat regions where the albedo is constant. They do not need a regularization term since the areas where the multi-view stereo approach fails, the shading-based term handles the problem. The usual surface representation for stereo algorithm is a depth value for each pixel [9] or a global point cloud [10]. In their work, Langguth et al. [4] use an approach proposed inside a multi-view framework by Semerjian [11]. Their approach uses bicubic patches to define a surface per view that has continuous depth and normals. In our work, we use a new approach based on a 3D Cartesian grid where we store a signed distance function. This allows us to have a representation that is also continuous for both depth and normals. We extend on their work by taking their output reconstruction as input mesh, converting it to a SDF and running our algorithm starting from there. We will show with some examples that our approach is able to reconstruct meshes starting from a simple sphere.

2.2 Differentiable rendering

While there is a lot of research around 3D reconstruction, that is not the case for differentiable rendering. The only public framework that can be found, as far as we know, is OpenDR from M. Loper [1]. Their framework is written in Python, uses OpenGL for the rendering part and Chumpy for the automatic differentiation. It offers a very flexible and easy-to-use solution, but also has some limitations. It is not specified in the paper, but since it uses OpenGL for the rendering, we deduce that we are limited to use rasterisation as rendering algorithm. This is not suitable for our work because the geometry representation we want to use is based on signed distance function. To be able to render it using rasterisation, we would need to convert it into a triangle mesh, which would be both complicated and expensive to do. We also wanted to use ray tracing as our image creation algorithm, which is another limitation of OpenDR since it is not supported.
OpenDR was also not performing well on some of the tests we did with it, hence we decided to implement our own framework in C++. To get the derivatives after our rendering process, we implemented a simple and minimal library to do *reverse automatic differentiation*, which is particularly suitable for our work, where we go from a very large input space to a single scalar output.
The framework is designed to be flexible and easy to extend. The whole code, with some output results and examples can be found at `https://github.com/SRaimondi/DRDemo`.

2.3 Automatic differentiation

There is a lot of research and software available when looking for automatic differentiation. In our work we decided to provide our own implementation and tried to make it as simple as possible so it would be much easier for other people who don't know the topic to use and understand our code. The code is mainly based on the work of M. Bcker et al. [12]. We will not provide here an explanation in deep on how automatic differentiation works. Interested readers should refer to their book where an extensive explanation of the theory behind automatic differentiation is provided.

3 Challenges and proposed method

In this chapter we explain in details our geometry representation, how we render it, the energy formulation proposed, how we compute and use the gradient and the minimization technique adopted. We first provide an outline of the processing pipeline designed and will go into the details on the next sections. Note that using the output of SMVS as our initial SDF is not mandatory, but it generally gives our optimizer a better initialization and reduces the computational time. Later some examples are shown using this approach.

3.1 Pipeline overview

Figure 3.1 shows the full pipeline of our system. The input images are initially passed to SMVS which produces a first reconstruction of the model. We assume our input model to be watertight in order to generate a SDF without open boundaries. This can be achieved by Poisson surface reconstruction [13] with Neumann boundary conditions. We use Meshlab on the output of SMVS for this and the result is fed into SDFGen, a program by Christopher Batty (`https://github.com/christopherbatty/SDFGen`) to convert a triangle mesh to SDF. At this stage both the target images and the generated SDF to initialise our grid for the minimization process are passed to our system. We have only done tests on virtual images up to now. This means that the tests we have done so far use target images that have been generated by our rendering system. We are currently limited to using this kind of images because in our algorithm we are not minimizing for the surface albedo nor the light parameters. Once our system finishes the minimization, the output is a file with the SDF values, the bounding box and the grid resolution. To be able to work on the mesh and compute the error with respect to the ground truth, we need to convert our representation into triangle mesh. We implemented a simple utility program based on the library *libigl* [14] which uses their implementation of the famous *Marching Cubes* [15] algorithm to extract a triangle mesh. The system also produces the final rendering of our SDF and a list of images at each iteration of the minimization from a single view.

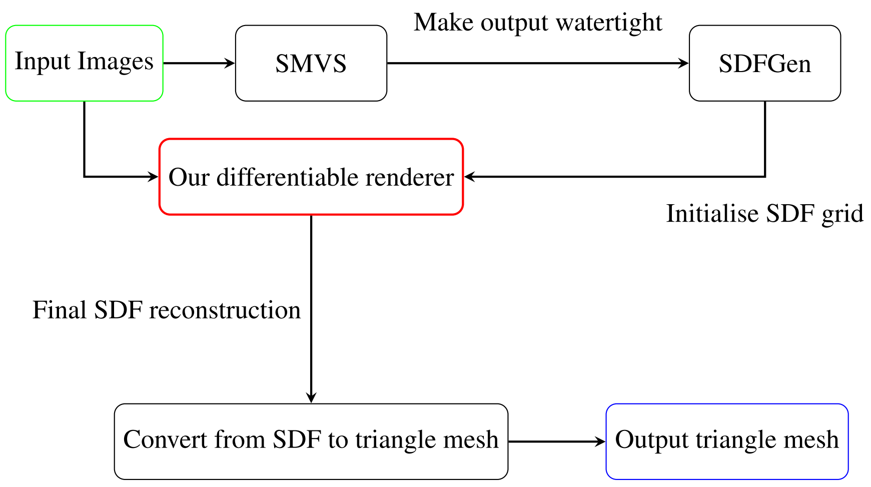

Figure 3.1: Full system pipeline. Input images are fed to SMVS to produce an initial guess of our model. We then use the input images as target to minimize the energy and finally output a triangle mesh of the reconstruction.

As already mentioned, the system is also able to work without an explicit initialization. Figure 3.2 shows the reduced pipeline in this case where the part using SMVS and SDFGen is skipped and we just work with the input images. The SDF is initialised using eq. (3.1) with a given radius.

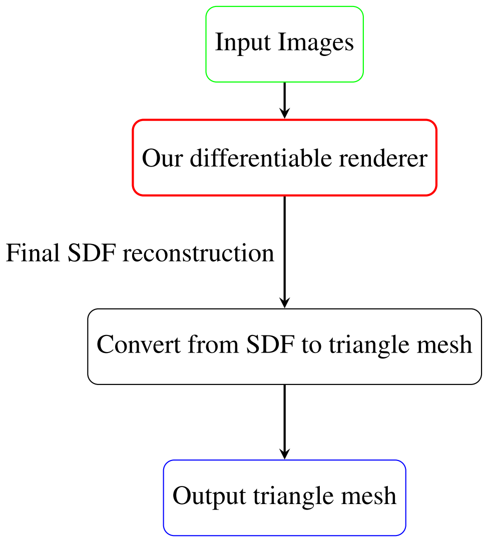
Input Images
Our differentiable renderer
Final SDF reconstruction
Convert from SDF to triangle mesh
Output triangle mesh

Figure 3.2: Reduced system pipeline. Input images are fed to our renderer directly. The output is then converted into a triangle mesh.

3.2 SDF representation

3.2.1 Data storage

As introduced in chapter 1, we decided to use a signed distance function to represent geometry for the reconstruction. We decided to use this representation especially because it offers a higher level of freedom for topological changes compared to triangle meshes. It is also able to capture small details in the geometry by simply increasing the resolution of the grid and is suitable for rendering. We assume that the represented mesh is always closed (watertight mesh). The straightforward way to store it is a dense 3D Cartesian grid and we store the value of our SDF at each crossing point. The grid is delimited by a fixed bounding box. This kind of representation is very memory inefficient since we might use a lot space to store non essential values of our SDF. In fact, we could also store only the voxels that contains the zero-isosurface since it is where our surface is. However, this would make handling the grid much more complicated. For this reason we decided to postpone an improved implementation for future work.
From this representation, it is trivial to get the value of our SDF at any point we want inside of its bounding box. In fact, we just need to compute the indices of the eight points of the voxel we are in and use triliner interpolation. This makes the level of detail achieved directly proportional to the resolution of the grid.
Another advantage of this representation is the simplicity of increasing the resolution of the grid. Our approach is based on the observation that SDF geometry representation is able to adapt well to a given target geometry by starting with a low resolution and increasing it gradually. In our test, the SDF is initialized using the following formula

$$\phi(\vec{x}) = |\vec{x}| - r. \tag{3.1}$$

This is one of the simplest SDF we can have, and it represents the distance from a sphere of radius r. In most of our tests, we use $r = 1$.
We can also initialise our SDF from a 3D triangle mesh description file like Wavefront after converting it using *SDFGen*.

3.2.2 Normals computation

Computing normals properly is very important for rendering since most of the shading models depend on it. We restrict ourself to a Lambertian shading model (more on this later), but since we are trying to reconstruct our geometry from images, it is crucial that the normals computed from our grid structure are precise. Moreover, recalling eq. (1.4), we know that the norm of the gradient $\nabla\phi$ should already be one at every point. To compute the gradient in the 3D Cartesian grid we decided to use the finite difference method. The formula we use is the second order central difference

$$\frac{\partial \phi_{i,j,k}}{\partial x} = \frac{\phi_{i+1,j,k} - \phi_{i-1,j,k}}{2\delta x} \tag{3.2}$$

where δx is the dimension of the voxel along the x axis. This formula can not be used when we are at the boundaries of the grid, so there we use the forward or backward, depending at which boundary we are, second order finite difference

$$\frac{\partial \phi_{i,j,k}}{\partial x} = \frac{-\frac{3}{2}\phi_{i,j,k} + 2\phi_{i+1,j,k} - \frac{1}{2}\phi_{i+2,j,k}}{\delta x}. \tag{3.3}$$

The derivatives along y and z are found in a similar way.
Initially, we used the first-order finite difference formulas but we noticed that using second order gave better results. One could argue that central difference does not involve the value at the point where we actually are. From our experiments, we did not notice any problem in using this approximation for the normal computation.

By using eq. (3.2) or eq. (3.3) at the boundaries, we are able to compute the gradient $\nabla\phi_{i,j,k}$ at each vertex of the voxel surrounding the point where we are. Once we have our eight normals we can again use triliner interpolation to compute the final gradient $\nabla\phi_{\vec{x}}$ at the given 3D point. Although the gradient should already satisfy eq. (1.4), we normalise it when a ray hits. The final formula of the normal at an intersection between a ray and the SDF is

$$\vec{n} = \frac{\nabla\phi}{|\nabla\phi|}. \tag{3.4}$$

This step is crucial because it does not allow the minimisation process to ”cheat” and match the target images by changing the SDF value in a way that the shading looks correct, but the geometry is not. This is one of the downside of our representation compared to triangle meshes, where this problem is not present.

3.2.3 Multi-scale

As we introduced before, our approach is based on progressively increasing the resolution of the SDF storage grid. This allows to first create a rough approximation of the target geometry and then refining to capture the small details. This behaviour will become clearer when we will present some real examples of our algorithm.
We explained in the previous section 3.2.1 that our SDF is stored in a dense grid. To increase the resolution, we use a very straightforward approach: we create a new grid with the same bounding box but with a smaller step between the sampling points. Now, we can simply go through all of the new points and use trilinear interpolation on the previous grid to get the new values. This approach is simple, robust and fast. Although a more sophisticated one could lead to better result, our results show that this method works well in practice.

3.3 SDF rendering

The usual way a ray tracer works is to find the closest intersecting object, if any, and then compute the shading for that point. Our implementation is not different, and since we wanted to keep the code generic, we had to find a way to provide back the same intersection information from our SDF as for a triangle. Fortunately, SDF can be easily rendered with the ray marching algorithm.

3.3.1 Ray marching

As we just explained, ray tracing finds the intersection point, if any exists, on a ray by using a ray-surface intersection routine. In our case, we can not directly compute where the ray will intersect our SDF, or if it will miss. The ray marching algorithm solves exactly this problem. The concept behind ray marching is that, starting from the ray origin, we compute a maximum distance estimate from the surface. In other words, we want to know how far we are from the closest point on the surface. Once we have that value we know that we can move along this ray, by using a normalized direction, as far as the value we computed. This process is show in fig. 3.3, where we can see the points in dark green and the distance estimation in light green. Every time the shortest distance to the scene (in black) is computed, we can advance along the ray by that distance and we know we are going to be at most on the surface itself. The process stops after a fixed number of iterations has been reached, in case we don’t hit anything, or if the distance to the surface is small enough. In that case, we have a hit with the surface and we can get back the intersection information.

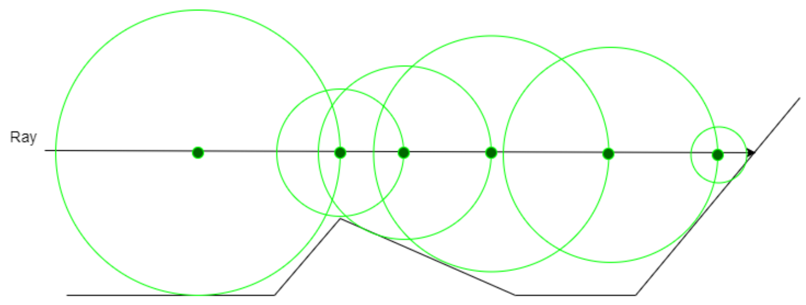

Figure 3.3: Ray marching algorithm. At each point, the closest distance to our mesh is computed and we can advance along the ray for that much. In this way we know that we will at most end up precisely on the surface.

Since we are using a SDF, it is trivial to use the ray marching algorithm to render our geometry. In fact, we just need to compute the distance to the surrounding bounding box if the point we are testing is outside. If we are inside, we can use the method explained above with triliner interpolation to compute the value at our testing point and then advance along our ray using that returned value. This also makes the rendering of our SDF very fast since we don't need to traverse each voxel that is intersected by the ray and see if the surface is in that particular one. We simply travel by the returned distance and stop when we are close enough to the surface. If we have a hit, the normal is computed as explained before eq. (3.4).

3.3.2 Lighting and shading

Keeping in mind that we wanted to keep our code as generic as possible to allow people to work on it, we provide a generic light interface. The integrator class in our code, which is responsible for computing incoming radiance at a given intersection, does not make any particular assumption on the type of light we are using. In the next three sections we explain the two light models already provided and how the shading is computed. This theory would help us to better understand the results we present later.

3.3.3 Direct light

The first light model is based on the camera looking direction. This is how the lighting is computed when no lights are added to the scene description. We can describe how the incoming light is computed using a Dirac delta function. Suppose the camera is looking at the scene along direction $-\vec{w}_c$. We can define the incoming light at a certain point as

$$L_i(\vec{w}_i) = \delta\left(\vec{w}_i - \vec{w}_c\right) \tag{3.5}$$

where $L_i(\vec{w}_i)$ is the light coming from direction $\vec{w}_i$. The minus sign in the camera looking direction definition is there to respect the convention used in ray tracing that incoming and outgoing light directions are always defined to point away from the surface. A more practical explanation of this light model would be to imagine the light as an infinite far away point light and all visible points receive light from a single direction. Basically the incoming light value is 1 when $\vec{w}_i = \vec{w}_c$ and 0 otherwise. Since the focus of our work was mostly on geometry representation, we use this model for a lot of tests, but we also investigated a more advanced representation based on spherical harmonics, which is the subject of the next section.

3.3.4 Spherical harmonics light

Spherical harmonics are functions defined on the surface of a sphere. They are widely used in science and also in computer graphics for global light maps representation. We added this light model and did one test

with it to give a direction in which the work should further expand. Our implementation is based on the work of Robin Green [16].
Spherical harmonics work well with our shading model, described in the next section, because they are smooth for Lambertian surface reflectance. In our test we use SH with 4 bands which gives us 16 coefficients for the light representation. We only tested with artificial light configuration initialised from a spherical function of the form $f\ (\phi, \theta)$, where ϕ is the rotation on the horizontal plane around the origin and θ is the rotation with respect to the vertical axis. Although we do not minimize for the light in our current version of the algorithm, we added this light model with a glimpse at the future. In fact, since spherical harmonics are good at capturing an approximation of the global lighting, in a next iteration of our model, we would like to include the spherical harmonics coefficients to be able to also reconstruct the light condition found in the target images and not only the geometry.

3.3.5 Shading model

The shading is based on some very simple assumptions. We fix the albedo of the surface to be equal to 1 and use the Lambertian reflection model. Dealing with varying albedo and especially with more complicated shading model is one of the biggest challenge in 3D reconstruction. In fact, also state of the art method like SMVS still assumes Lambertian reflectance. Trying to work with different reflection models was one of the possible goals of the work, but we decided to focus more on the geometry part and leave the usage of more advanced shading models for the future.
Putting what we have described before for the lighting model and the shading model, we can formulate the rendering equation we use in our system for the moment as

$$L_o(\vec{w}_o) = \rho \int_{\vec{w}_i} L_i(\vec{w}_i) max(0, \vec{n} \cdot \vec{w}_i) d\vec{w}_i \tag{3.6}$$

where

- $L_o(\vec{w}_o)$ is the outgoing light in direction $\vec{w}_o$
- ρ is the surface albedo, 1 in our case
- $L_i(\vec{w}_i)$ is the light coming from direction $\vec{w}_i$

Later we will see though that our system does not make any assumption on how we are doing the rendering, eq. (3.6) is given so that the reader can have a better idea of what we did in the tests and understand the results.

3.4 Energy formulation

In this section, we go through the different attempts we did to formulate a minimization objective that would work with our geometry representation. We assume that lighting and camera position are fixed so we only have to minimize our energy with respect to the geometry.

3.4.1 Initial formulation

The way we formulate our minimization problem is crucial for having a good final reconstruction of our target geometry. Here we will go through the different attempts we made and the final energy formulation we propose.
Recalling what we explained in section 1.2, our final goal is to match the image produced by our forward

rendering process to a target one. In our case we have multiple images and for each view we want to match it as close as possible. We can reformulate eq. (1.2) to include multiple images in the following way

$$E_I = \sum_t |f(\Theta_t) - I_t|^2 \tag{3.7}$$

where t is the index of the target image and Θ_t are the parameters of our forward rendering function with the camera in the position from where we expect to get the target image I_t. The initial attempt to use this energy turned out to be unsuccessful. We were getting very poor results, but we quickly found out what the problem was. Most of the values in the energy gradient were zeros. This was due to the fact that a lot of the points in our 3D Cartesian grid where not involved in the SDF rendering, so we where not able to find the derivatives for those points through automatic differentiation of our renderer.

3.4.2 The reinitialization equation

The next attempt was inspired by the work of Sussman et al. [17]. Here we did not change our energy formulation but instead investigated into the *reinitialization equation*

$$\frac{\partial \phi}{\partial t} + S(\phi)(|\nabla \phi| - 1) = 0. \tag{3.8}$$

This partial differential tries to make the value of the norm of the gradient equals to one at each point. If this condition is satisfied, then $(|\nabla \phi| - 1)$ becomes zero and the derivative with respect to time vanishes, meaning we are at a steady state. The term $S(\phi)$ also plays a key role. In the basic formulation, $S(\phi)$ is a sign function that takes value 1 in Ω^+, -1 in Ω^- and 0 on $\partial\Omega$, where we want ϕ to stay identically equal to zero. Unfortunately, if ϕ is not smooth or if the gradient has some large value in it, we might get into the situation where our interface moves incorrectly, and that's not what we want.
The attempt was to try to solve this equation on our SDF after doing one minimization step using the gradient we got back from eq. (3.7). We implemented an algorithm to solve eq. (3.8) based on Godunov's scheme, as suggested in [8]. Unfortunately, this approach was not successful. The SDF was degenerating after each step. We then tried to change the $S(\phi)$ term as suggested by Peng et al. [18]. The formula they propose is

$$S(\phi) = \frac{\phi}{\sqrt{\phi^2 + |\nabla \phi|^2 (\delta x)^2}}. \tag{3.9}$$

This new sign function smears out the value of the sign slowing down his propagation. Equation (3.9) is particularly effective when the initial value of ϕ is a poor estimate, which is exactly our situation after the gradient propagation step. Even with this new solution, the results we got were very miserable, the fundamental condition of the SDF eq. (1.4) was not been respected at all during the minimization process and also the reconstruction was very unsatisfying. Moreover, the computation time to solve eq. (3.8) using eq. (3.9) was becoming considerably high. In fact, eq. (3.9) needs to be updated at every iteration step. After all the attempts we did using this approach and failing, we started to investigate if it was possible to find a new energy formulation that would automatically provide us with a gradient for all the points in our grid, without the need for additional steps.

3.4.3 Regularization term on normals

Recalling eq. (1.4), our intuition was to add a regularization term on the energy that used this condition to keep the geometry representation a proper signed distance function.
We formulated an energy term over the normals of our SDF as

$$E_{\nabla \phi} = \sum_{i,j,k} \left(|\nabla \phi_{i,j,k}|^2 - 1 \right)^2 \tag{3.10}$$

where $\nabla\phi_{i,j,k}$ is computed as explained in section 3.2.2.
The idea of this term is that, as long as the length of the normal at each point is close to 1, the energy is going to be minimized when our stored SDF satisfies eq. (1.4) as much as possible. We added a square on the term so that, even if the length of the gradient is smaller than one, we have a sum of positive terms.

3.4.4 Final energy formulation

Following our initial energy formulation in eq. (3.7) and eq. (3.10), we define the final minimization objective as

$$E = E_I + E_{\nabla\phi} = \sum_t |f(\Theta_t) - I_t|^2 + \lambda \sum_{i,j,k} \left(|\nabla\phi_{i,j,k}|^2 - 1 \right)^2 . \tag{3.11}$$

Here λ regulates how much importance we give to the regularization term. We can see from eq. (3.11) that it is not possible to compute a direct solution for the gradient since we have no information at all about $f(\Theta_i)$. This is the reason why we rely on the automatic differentiation library to do the heavy lifting for us and provide the gradient of the minimisation objective when needed.
Notice that the output of $f(\Theta_t)$ and the targets I_t could also be a normals map or depth map. We would just need to change the output of f in the rendering system.

3.5 Minimization

Minimizing a target energy function can be a complex task, especially when the number of variables becomes high and we do not have many knowledge on the target function, which is exactly our case. There are some methods that make use of the Hessian matrix to compute a better search direction and a more accurate step size (e.g. dogleg method). In our case, we do not have access to second order information about our function because the automatic differentiation library does not support it.
For this reason, we decided to test our method using *Backtracking Line Search Method* for the step length and to use gradient descent as minimization method. Although simple, the method works quite well. The downside of backtracking is that, if we want to carefully find the maximum step-length that decreases our function value, we need to test a lot of possibilities and since we have to render a few images each time, this becomes quite computationally expensive.
We already discussed the multi-scale approach and here we will go a bit more in depth into it. We found out by experimenting that starting from a low resolution 3D Cartesian grid and increasing it slowly leads to good results. The approach is quite simple: suppose the grid starts with a given resolution $(n_x \times n_y \times n_z)$, we give a factor $s > 1$ and define manually a number of refinement steps. Each time the minimisation at a certain resolution ends, we increase it to $(s \cdot n_x \times s \cdot n_y \times s \cdot n_z)$. We then compute the values of our SDF with trilinear interpolation on the previous grid for the new one and start the minimisation again. This gives the algorithm the freedom to first adapt the SDF roughly to the visual hull of the target mesh. Then, by increasing the resolution, we are able to capture smaller and smaller details. As one can imagine, the value we set for s plays a crucial role in the output. We did not have time to find a general and robust way of setting s adaptively at each step. Future work on this would be to find some way to relate the error on the images to choose s adaptively and not by using a fixed value. From our experiments, depending if we are trying to reconstruct a target mesh from a very different geometry, a value of s from 1.5 to 2 works well. If we are focusing on reconstructing details, a value of $s \approx 1.2$ is better.
Another important role is played by the starting grid resolution. If we are able to initialise our SDF with a good approximation through the mesh visual hull or the output of SMVS, then we can already use a high resolution as starting point. Otherwise, the algorithm will not be able to adapt to the big geometrical changes needed to match the target model. In algorithm 1 an outline of how the whole process works is outlined to give a better view on it. We also briefly explain how the minimization step works. The criteria

we use to stop are either the length of the gradient is smaller than a given tolerance or if the step length α becomes too small ($\approx 10^{-12}$). The gradient ∇E_Ω is the gradient of E with respect to all the values of our SDF Ω. Therefore, the number of variables we minimize for is $n_x \cdot n_y \cdot n_z$, which can quickly become very large.

Algorithm 1 Minimize energy E given $SDF\ \Omega$

```
Require: $s > 1, N \geq 0$                         ▷ N: number of refinements done on the grid
 1: Initialise $\Omega$
 2: MINIMIZE($E, \Omega$)
 3: $i \leftarrow 0$
 4: while $i < N$ do
 5:     Get $\Omega$ resolution $(n_x, n_y, n_z)$
 6:     Create new grid $\Omega_{new}$ with resolution $(s \cdot n_x, s \cdot n_y, s \cdot n_z)$
 7:     Interpolate on $\Omega$ to get $\Omega_{new}$
 8:     $\Omega \leftarrow \Omega_{new}$
 9:     MINIMIZE($E, \Omega$)
10:     $i \leftarrow i + 1$
11: end while
12:
13: function MINIMIZE($E, \Omega$)
14:     while stop conditions for minimization are not satisfied do
15:         Evaluate $E$ and compute $\nabla E_\Omega$
16:         Find best step length $\alpha$ along search direction $-\nabla E_\Omega$      ▷ Backtracking method
17:         Update $\Omega \leftarrow \Omega - \alpha \nabla E_\Omega$
18:     end while
19: end function
```

4 Results

In this chapter we show how our algorithm performs in different situations. First a few test using the direct light model are presented. After, we illustrate what happens using the spherical harmonics light model. In the last section we present a failure case where we tried to use real object images. On the project Github page, you can find all the images at full resolution. In each folder, a sequence of images showing the rendering of our SDF for the first view at each iteration is also available.

4.1 Direct light

4.1.1 Cube

The first test we present is the recreation of a cube starting from a sphere. Below, in fig. 4.1 we show 4 of the 8 views used as target view to reconstruct the geometry. The views look very similar because the camera positions are placed around the cube while moving up and down. The direct lighting model also contributes to make the image all similar. We use a resolution of 256×256 for this test.

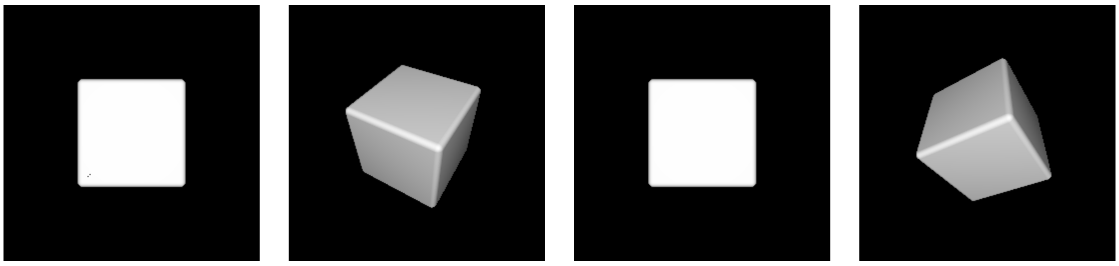

Figure 4.1: 4 views of the 8 used in the cube reconstruction test. We only show 4 out of the 8 because the other 4 are equals since the camera positions are symmetric.

Next we show how our SDF looks like at the beginning of the minimization process in fig. 4.2.

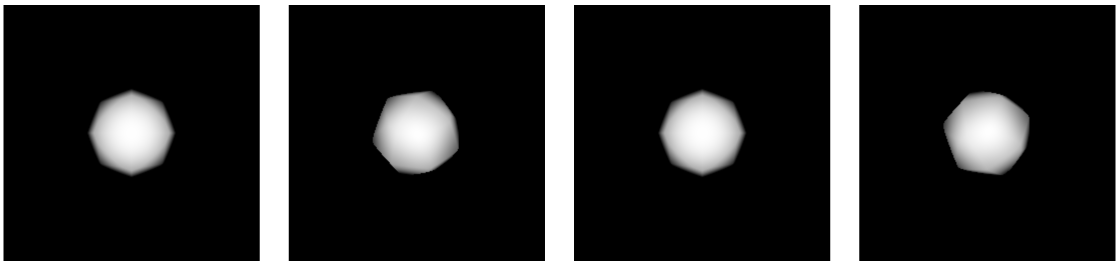

Figure 4.2: Initial rendering of our SDF, initialized with eq. (3.1) using $r = 1$.

In fig. 4.3 the same 4 views of the final rendering of our SDF are shown.

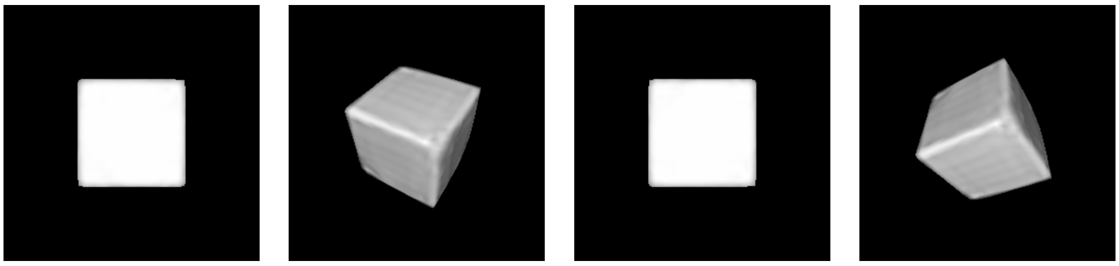

Figure 4.3: 4 views of the output 8 where we can see the rendering of our SDF after the minimization.

We can observe that the algorithm did a pretty good job at matching our input geometry starting from the sphere. We configured this test with an initial resolution of $7 \times 7 \times 7$ and we did 2 additional refinement steps, doubling the resolution each time. The value of λ is set to 1 for this test.

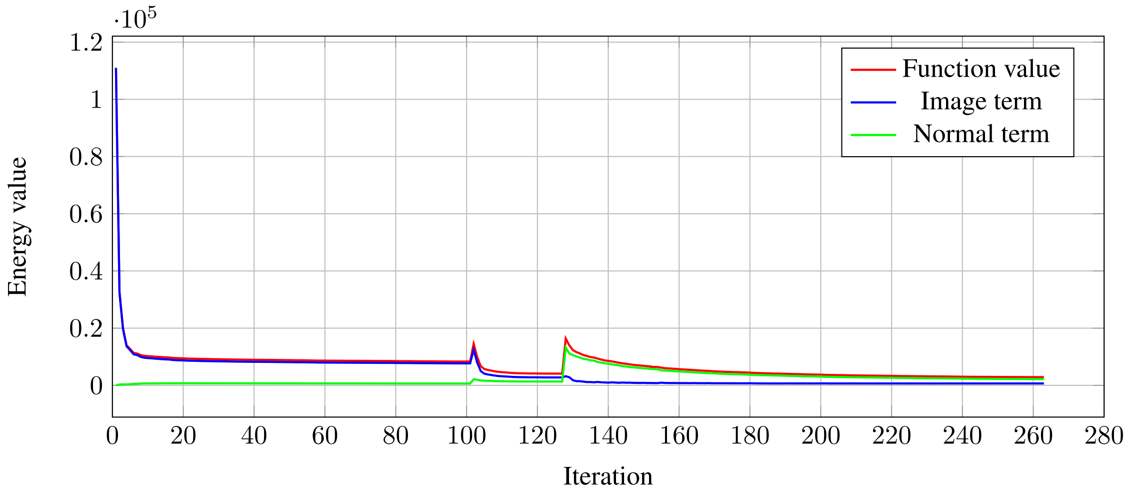

Figure 4.4: Plot of the energy value and the two single terms for the cube test.

In fig. 4.4, we show the plot of the energy during the minimisation with both the image and the normal term. We can observe how at the beginning of the minimization, the Image term is the dominating one since our geometry is very different compared to the target one. From the graph, it's clear that we get a very close guess of our geometry in a few iterations. In fact, already at iteration 20, the energy is more that five times smaller compared to the starting one. Another interesting behaviour we can observe on the graph is the presence of two peaks around iteration 100 and 130. This are the points where the resolution of our SDF grid was increased. In fact, what we can see on the graph is that at the first refinement the Image term increases. This comes from the interpolation scheme we use which produces small changes in the geometry when we refine it.
The normal term also shows some peaks around the refinement iterations. This behaviour is quite easy to understand. As we defined in eq. (3.10), our Normal term is computed as a sum of the length over all the points of the grid. If we increase the resolution, we will have more points at which the normal is computed. Since we are working in a discrete domain, it's not possible to have perfect normals with length 1, so we will have a small error at each grid point. By increasing the resolution, the error on the normals increases naturally. We can see however, that the minimization process is able to recover on the error introduced by the refinement. In fact, from iteration 130 on, the energy is dominated by the Normal term and the Image term goes close to 0 and remains constant. This behaviour will show up again in the other tests. All the images for the cube test, including the iterations from the first view, can be found at `https://github.com/SRaimondi/DRDemo/tree/master/output_final/cube`.

4.1.2 Sphere with noise

We test our algorithm, still using the reduced pipeline, in the reconstruction of a sphere with some smooth noise on top of it. Again, the lighting model used is the direct one. We also use 8 views for this test but we only show 4 of them here. The purpose of this test is to get a first idea of how well the algorithm deals with details on a target mesh. Figure 4.5 show 4 of the target views used for the test. The camera is configured as for the cube test section 4.1.1 but here we can clearly see the difference since the target mesh is not symmetric. We started with a resolution of $(10 \times 10 \times 10)$ and did 2 refinement steps, again doubling the resolution each time. The value of λ in the energy from eq. (3.11) is set to 1 and again the image resolution used is 256×256.

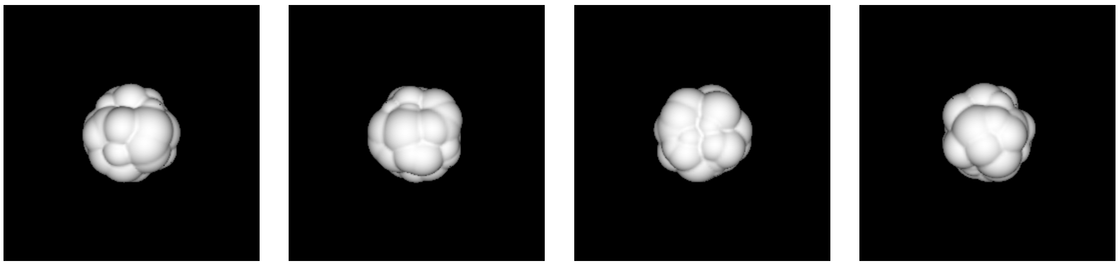

Figure 4.5: 4 views of the 8 used in the sphere with noise reconstruction test as target images.

Next we find the initial rendering of our SDF at the beginning of the minimization.

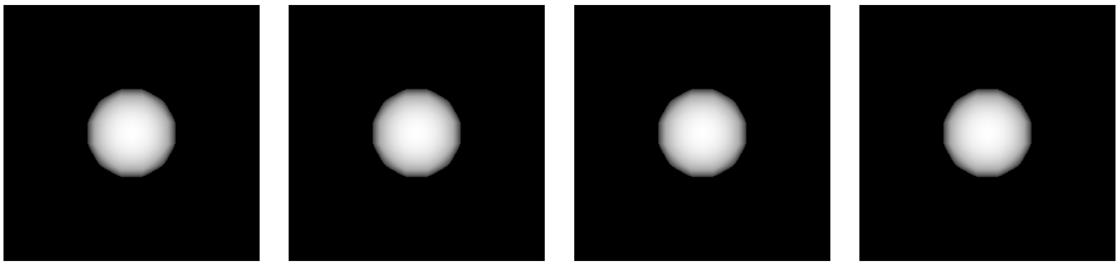

Figure 4.6: The starting rendering of our SDF for the 4 views shown before.

Last, the same 4 views are shown with the rendering of our SDF at the end of the process.

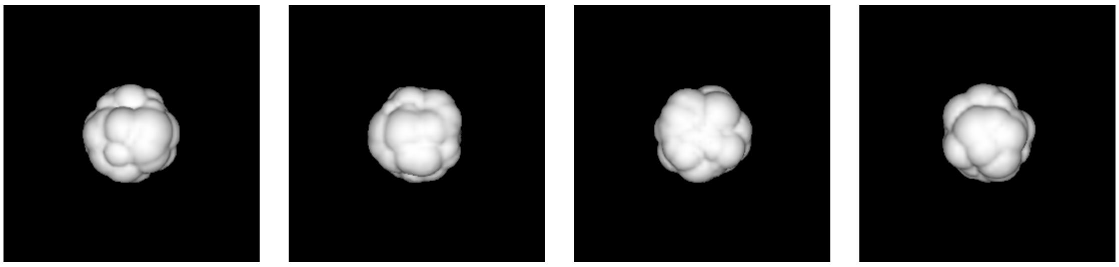

Figure 4.7: Final rendering of our SDF grid after minimization.

We can already see from the visual result in fig. 4.7 that the reconstruction of our target geometry worked out well. The energy graph gives an even better view on that.

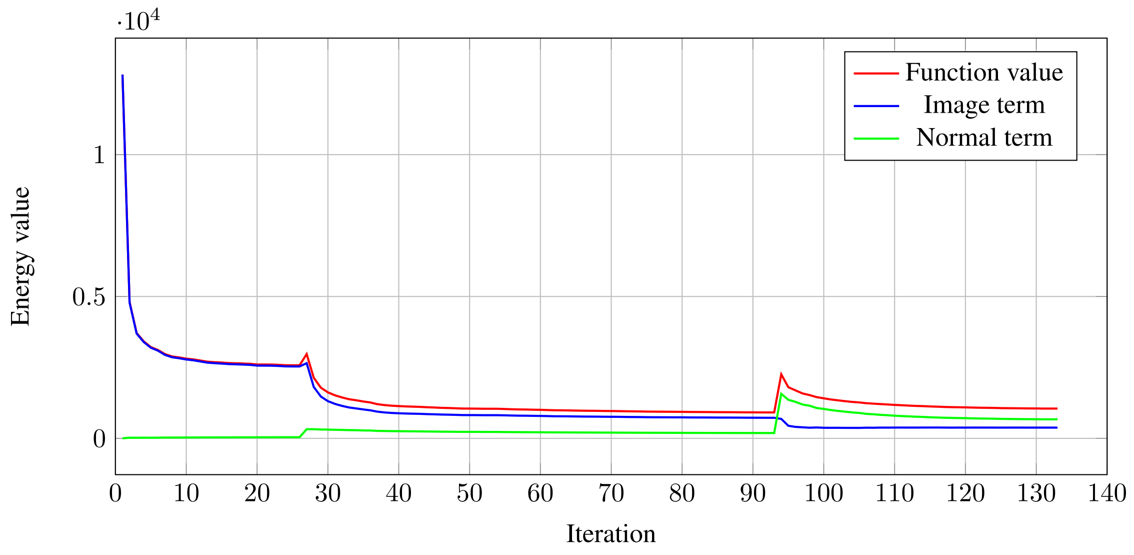

Figure 4.8: Plot of the energy value and the two single terms for the sphere with smooth noise test.

The behaviour in fig. 4.8 is similar to the one of fig. 4.4. At the beginning, we are able to greatly decrease the energy through the Image term. The Normal term presents the same behaviour as before, with the peaks at the refinement iterations and the correction done later by the minimisation process. In the last iterations, the Normal term becomes higher than the image term, due to the high number of points in the grid $(40 \times 40 \times 40)$, after the refinement. The algorithm is still able to correct it while also keeping the Image term low, which is what we want to match the target geometry as good as possible.
We report the *Hausdorff Distance filter* output from Meshlab in table 4.1 to give an idea of the final error on our mesh, after converting it into triangles, with respect to the ground truth we used for the rendering.

	Minimum	Maximum	Mean
Error	2e−6	0.0997	0.00921
Error w.r.t BBox diagonal	1e−6	0.0273	0.00252

Table 4.1: Hausdorff distance for our output triangle mesh compared to ground-truth. The first row reports the absolute value on the mesh. The second one is w.r.t to the ground truth's bounding box diagonal (≈ 3.6).

As we can see, the max and the mean error are very small, considering our target sphere had radius ≈ 1 when we applied our noise. This is even more obvious when looking at the second row of table 4.1 where the error is reported with respect to the bounding box diagonal. To give an even better idea of the quality of our output, we show two images below in fig. 4.9 of the target mesh coloured with the error made at that point. The gradient goes from red for very small error to blue where we have our maximum error.

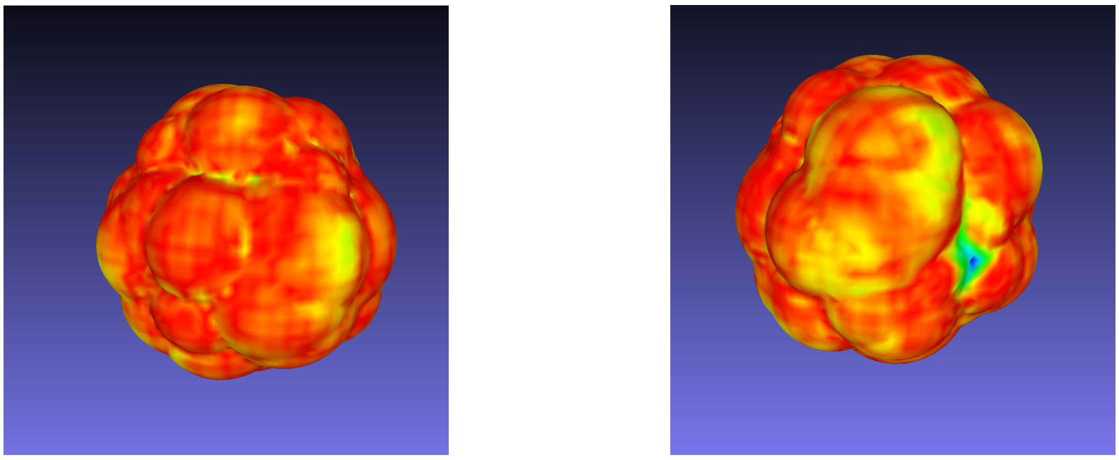

Figure 4.9: Error on our reconstruction compared to the the ground-truth model. The colour gradient goes from red (small error) to blue (big error).

Figure 4.9 shows how the reconstruction overall is very precise. There is only a small region on one side of the mesh, visible in the right image, where there is a small area that was not reconstructed properly and the error takes its maximum value. All the images are available at `https://github.com/SRaimondi/DRDemo/tree/master/output_final/blob_10_2`.

4.1.3 Stanford bunny

The next test shows what our system is able to do without an explicit initialization while trying to recreate a complex model. We target the famous Stanford bunny model as the geometry we want to reconstruct and initialize our SDF grid with a simple sphere of radius 1. The number of views used is 8 and the image resolution is 256×256. 4 of the target views are shown below in fig. 4.10.

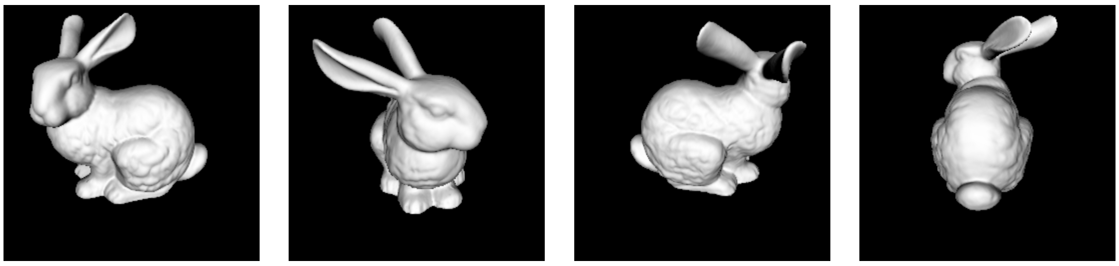

Figure 4.10: 4 of the 8 views used for the Stanford bunny test

Here we do not report the starting rendering of our SDF since it's the same (sphere initialization) as for the previous tests. Again, all images can be found in the Github page of the project at `https://github.com/SRaimondi/DRDemo/tree/master/output_final/bunny`.
The same 4 view with the final rendering of our SDF after the minimization are shown below in fig. 4.11. We set the initial resolution of the grid to be $10 \times 10 \times 10$ and do 6 additional refinement steps using a multiplier of 1.5. The final resolution is $109 \times 109 \times 109$.

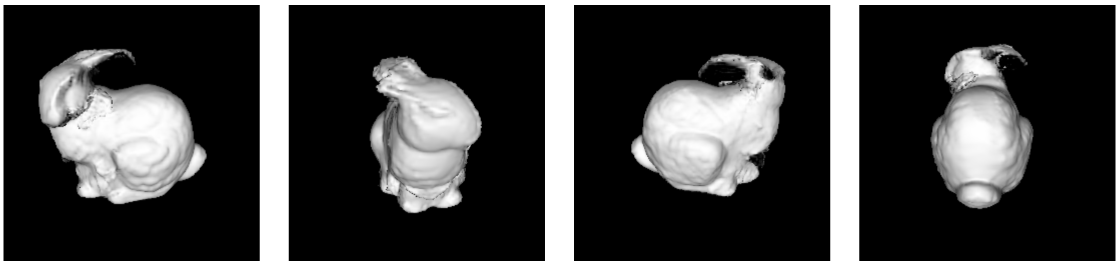

Figure 4.11: The final rendering of our SDF after the minimization for the same views used to show the target images in fig. 4.10.

Before commenting the results of this test, we expose the plot of the energy behaviour, the table with the quality of our reconstruction and some images of the error with respect to the ground-truth model.

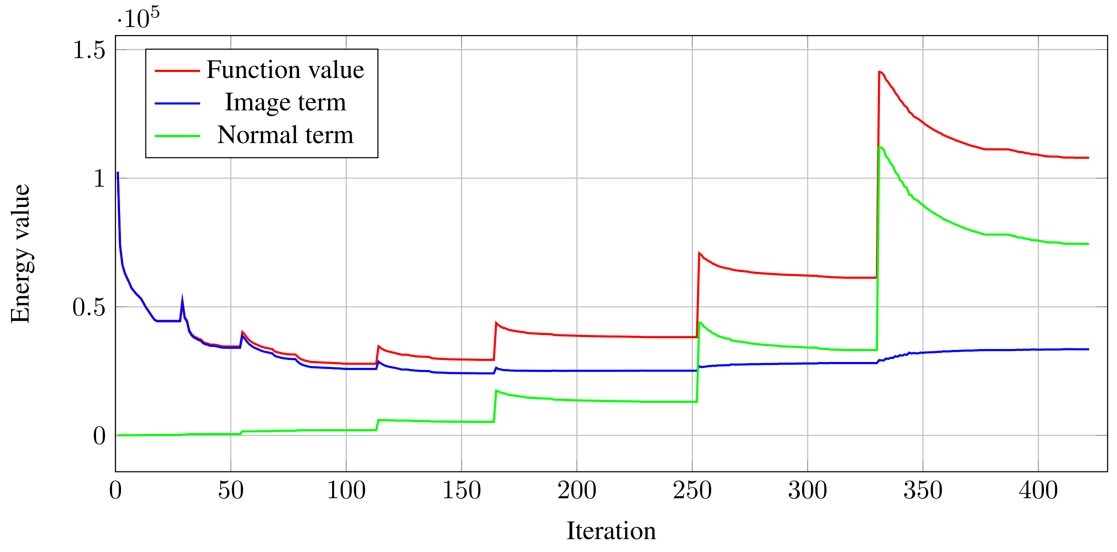

Figure 4.12: Plot of the energy value and the two single terms for the Stanford bunny test.

	Minimum	Maximum	Mean
Error	0	0.719	0.0627
Error w.r.t BBox diagonal	0	0.192	0.0167

Table 4.2: Hausdorff distance for our output triangle mesh compared to ground-truth for the Stanford Bunny test. The bounding box diagonal of the model used is ≈ 3.75.

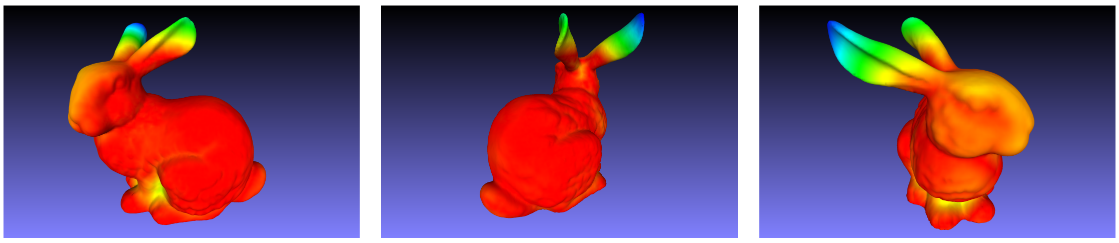

Figure 4.13: Error on our reconstruction compared to the the ground-truth model. The error increases when going from red to blue.

We can see from fig. 4.11 that our method did quite a good job at reconstructing the target geometry but there are some parts where it failed. As one can better observe from the images in fig. 4.13, the body of the bunny was reconstructed without problems and the error is small. The parts that our algorithm fails to reconstruct properly are the ears. In fact, the error is very high in this area, as shown in fig. 4.13 and we can also observe it in the final rendering of the SDF in fig. 4.11. Another region where the algorithm had some problems was the head of the bunny where the margin of error is higher than the body. It is clear that reconstructing the body was much easier for our algorithm because the change compared to the starting sphere was not as big. For the ears, we can see that the method tries to extrude from the head to recreate the ears but the reconstruction is very imprecise. It's also possible to observe some artifacts on our mesh

around the head in fig. 4.11.
Moving on to the graph of the energy in fig. 4.12, we can see how here the pattern changed compared to the previous tests. In fact, in the first iterations, the minimization works well and the energy value is reduced considerably together with the Image term, while the Normal term increases slightly but it's still very small. Also in this test, we can see the 6 peaks in the energy due to the refinement we did on our grid during the whole reconstruction process. It is interesting to see what happens after the fourth refinement step around iteration 160. From that point on, the Image term is constant, so we are clearly not improving the reconstruction at the image level. After that point, the function value grows back even bigger than where we started from. This is due to the peaks in the Normal term which becomes very large. Again, the value of this term is strongly conditioned by the number of points in the grid, so at each refinement step it grows bigger and bigger. The minimization process, in the end, is just working to force the grid to respect the SDF condition in eq. (1.4). It is clear that we used too much refinement steps for this test, and in fact after iteration ≈ 330, the Image term is even going back up, which is not what we want at all. Our guess is that the minimization is not able to improve the image term any more given the level of details we can capture in our grid, therefore, the minimization is trying to reduce our energy through the Normal term only. As mentioned before, one of the weak spot of our method is having to set by hand all the parameters and there is still no way for the algorithm to find the optimal number of steps and the appropriate next resolution for the grid.

4.1.4 Pearl dragon

In this test we use our system in its entirety. The target images are shown in fig. 4.14. These images are passed to the SMVS program and we get out a first reconstruction used to initialize our SDF grid as shown in fig. 4.15. The first thing we can observe comparing fig. 4.15 and fig. 4.14 is the loss of details in the output. The algorithm starts with a grid resolution of $104 \times 142 \times 92$ and does 4 steps using $s = 1.1$ as refinement multiplier, ending up with a $150 \times 206 \times 134$ grid. The value of λ was set to 1 and to better capture the small details on the dragon, the image resolution is set to 512×512.

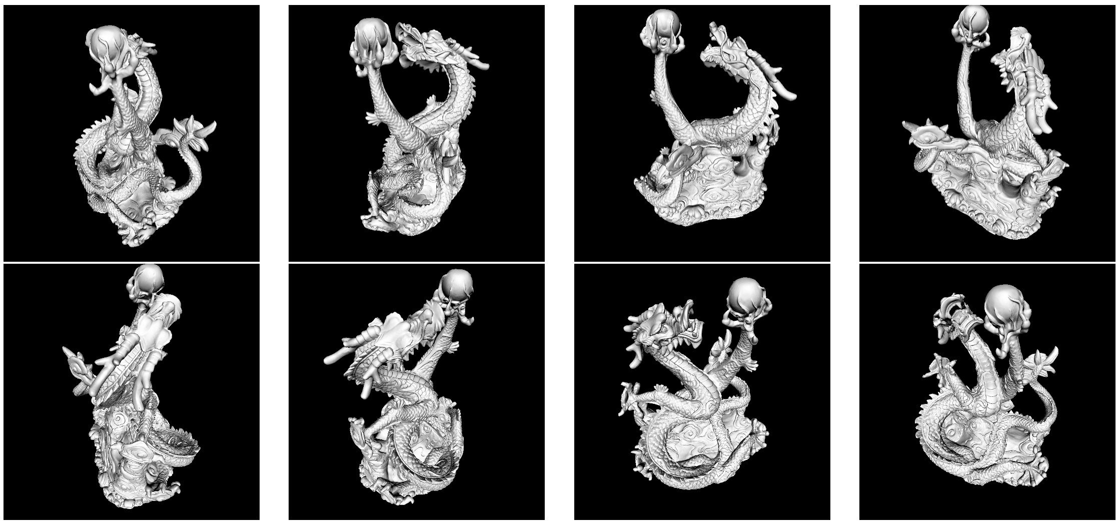
Figure 4.14: The 8 target views of the Pearl dragon model rendered with our system.

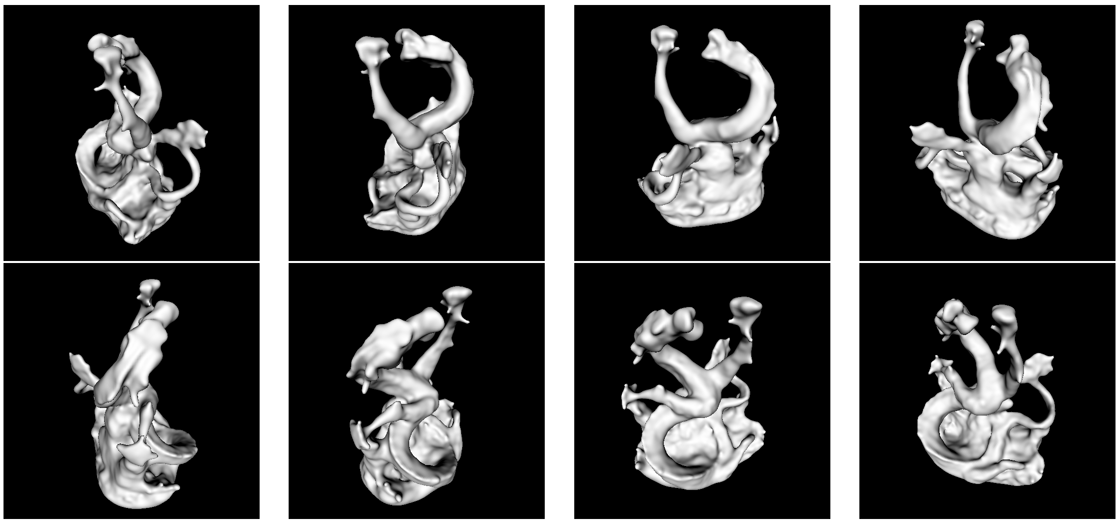
Figure 4.15: Initial rendering of our SDF grid initialised from the SMVS output.

After applying our algorithm, we get the output images in fig. 4.16. We present them in bigger size to better display how the details were recovered. All the images can be found at full resolution at `https://github.com/SRaimondi/DRDemo/tree/master/output_final/full_dragon`.

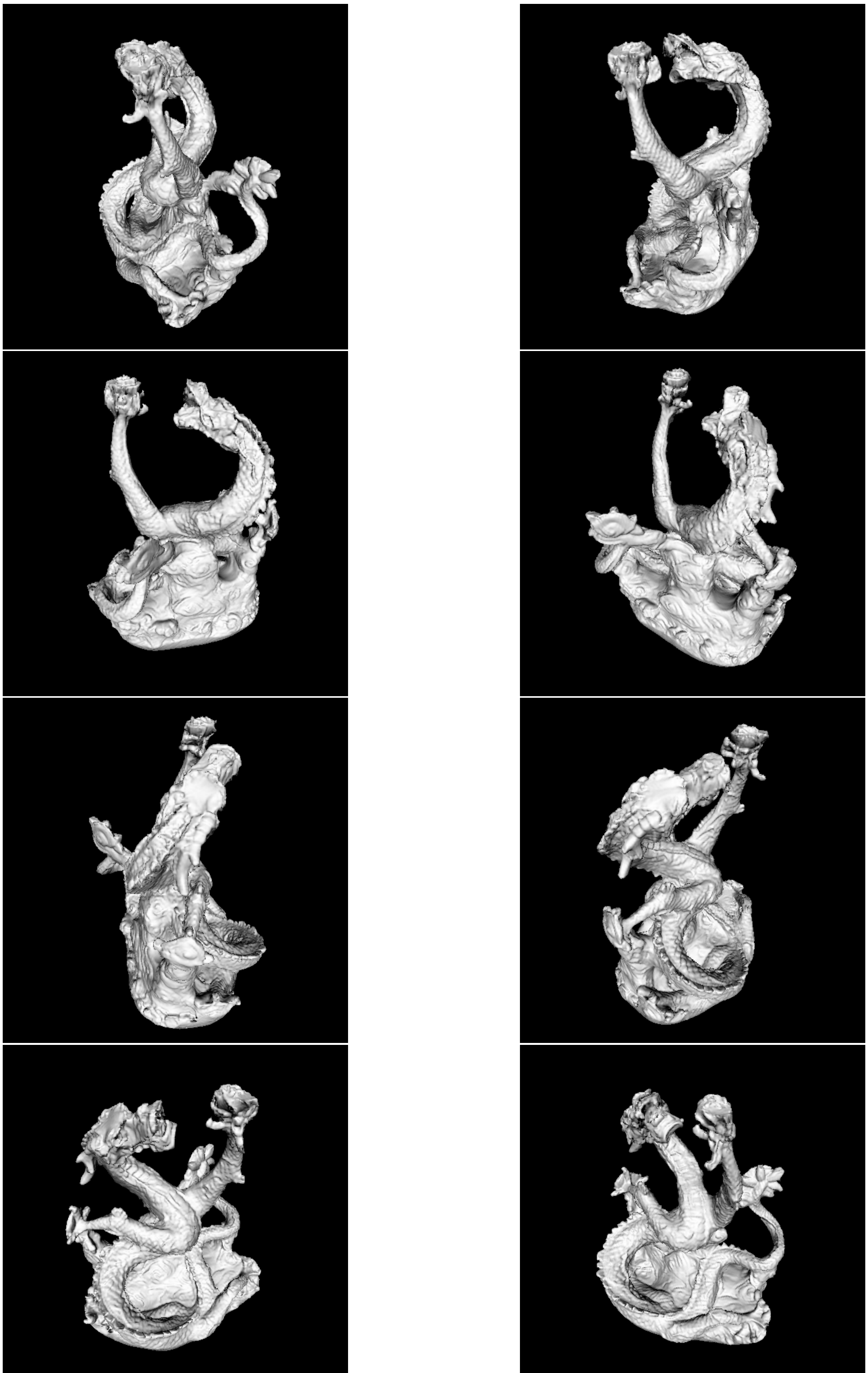

Figure 4.16: final rendering of our SDF grid after energy minimization.

Below, the graph with the energy behaviour for this problem is presented.

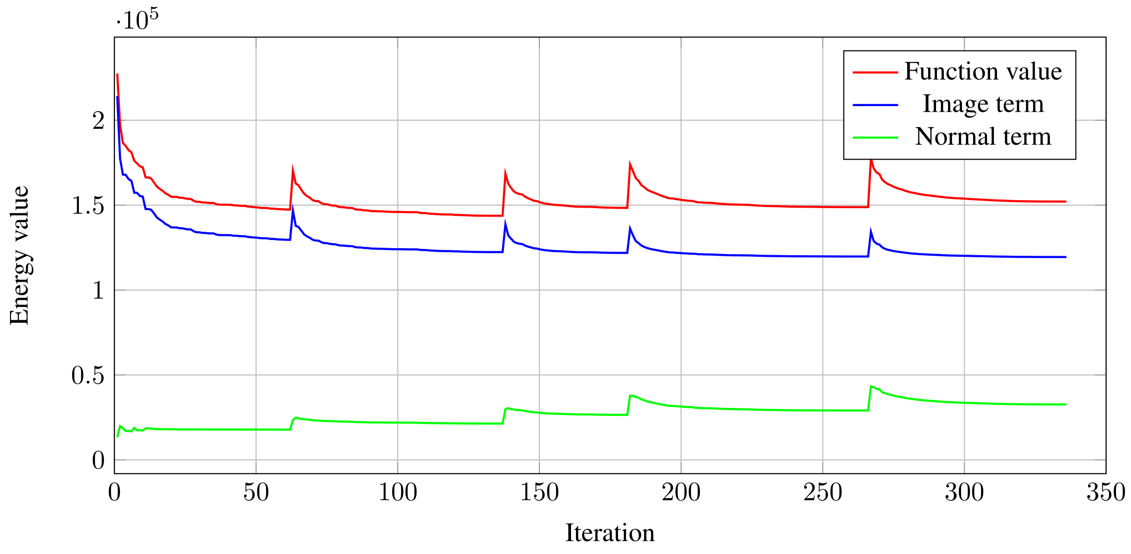

Figure 4.17: Plot of the energy value and the two single terms for the Pearl dragon test.

What can be detected in fig. 4.17 is that for this test the Image term is always the dominant one. We can remark how at the beginning we were able to reduce the energy considerably in a few iterations. After that, the pattern we already observed before with the spikes in the energy after the refinements is still present. The Normal term tends to grow during the minimization. The algorithm tries to readjust it after the refinement peaks but on the long run, the value of the term still increases. This again is mostly due to the small numerical errors we make at each grid point since we are working with finite differences to compute the derivatives.
The Image term shows an interesting behaviour which is worth discussing. At the beginning we succeed in reducing it considerably in a few iterations. If we look at fig. 4.15, we can see how the details of the model are all missing. Out method is able to recover them gradually. Especially on the skin of the dragon, results in fig. 4.16 clearly show the small details on the skin while the output of SMVS was very smooth. The problem is, as described before, that our method does not deal well with big geometrical changes if the resolution is too high to capture them. This is exactly what is happening here. A very clear example is the pearl at the top of the model. In fact, the output of SMVS is not capturing the spherical shape and since we start from a high resolution for the grid, we are not able to correctly reconstruct it. As one can see in our output images in fig. 4.16, the pearl is not reconstructed properly. There are other parts of the dragon showing this behaviour but this one is the most relevant and easy to see. These big errors already present in the input causes the energy to be very high at the end, because we are not able to make big changes in the geometry that would allow us to reduce it further.
We also compared the output model of SMVS and ours using the *Hausdorff Distance filter* to see the error on the reconstruction after our processing. The results are reported in table 4.3.

	Minimum	Maximum	Mean
Error of SMVS	0	0.466	0.0439
SMVS error w.r.t BBox diagonal	0	0.0847	0.0078
Error after our processing	0	0.466	0.0387
Our error w.r.t BBox diagonal	0	0.0845	0.007

Table 4.3: Hausdorff distance on the SMVS output and after our processing. The diagonal of the bounding box for this particular model is ≈ 5.5.

As we can see from the results in table 4.3, we were able to slightly reduce the mean error on the whole model by 12%. The maximum error did not change because we have some big differences caused by the lack of views on the bottom part of the model.
Below we present, for two views only, the absolute value of the difference at the image level. On the right column we find the difference between the initial rendering of the SDF, initialised from the SMVS output, and the target image. The left column displays the difference between the target image and the final rendering of our output SDF.

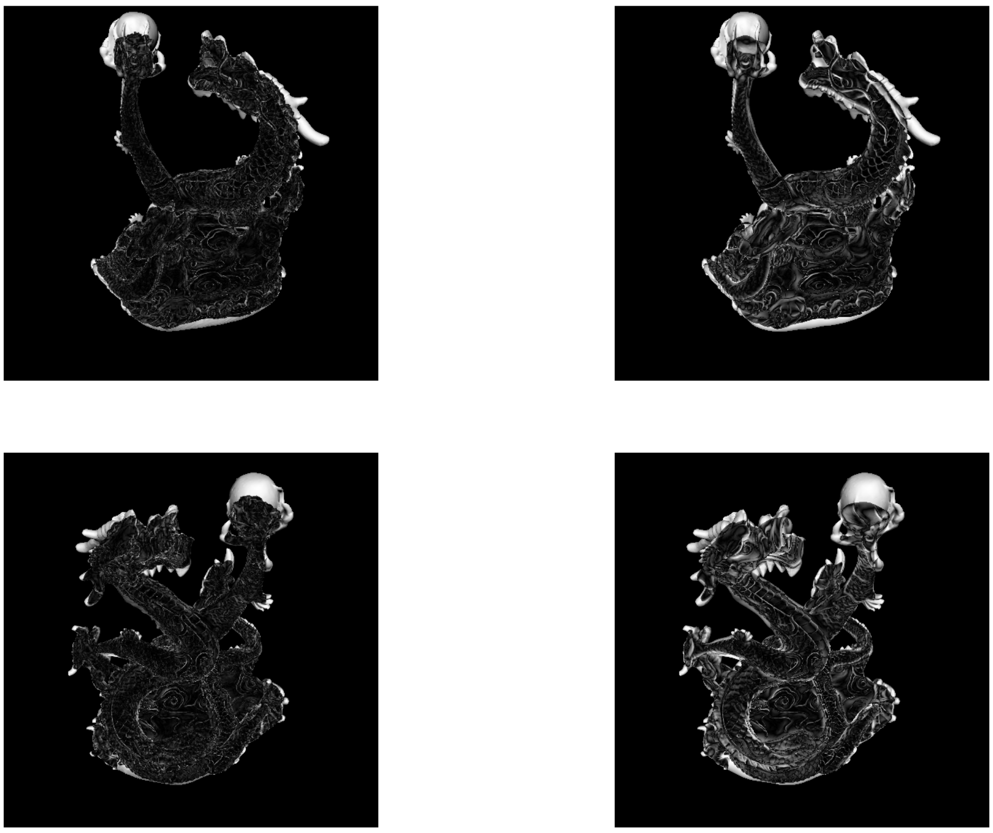

Figure 4.18: Left: difference between our final output image and the target one. Right: difference between our starting SDF rendering and target image.

Figure 4.18 helps to clarify even better the discussion we made before for the energy behaviour. By showing the images difference, it's possible to observe from the left column how there are some elements that have not been reconstructed at all (like the pearl or some parts of the head). What is notable is what happened on the body of the dragon. For both rows, we can clearly see how the difference in the images was greatly reduced at the small details level. In fact, if we take a look at the left column, we can see how there are some regions where the difference at the image level has become almost zero (black). Many of the grey areas on the body of the dragon, regions where the approximation was not good, on the right column have turned black when looking at the respective position in the left image. This means that our algorithm did an efficient job at minimizing the difference at the image level where possible.
Last, we show the mistake made on our reconstruction, compared to the ground-truth model, after converting it to a triangle mesh.

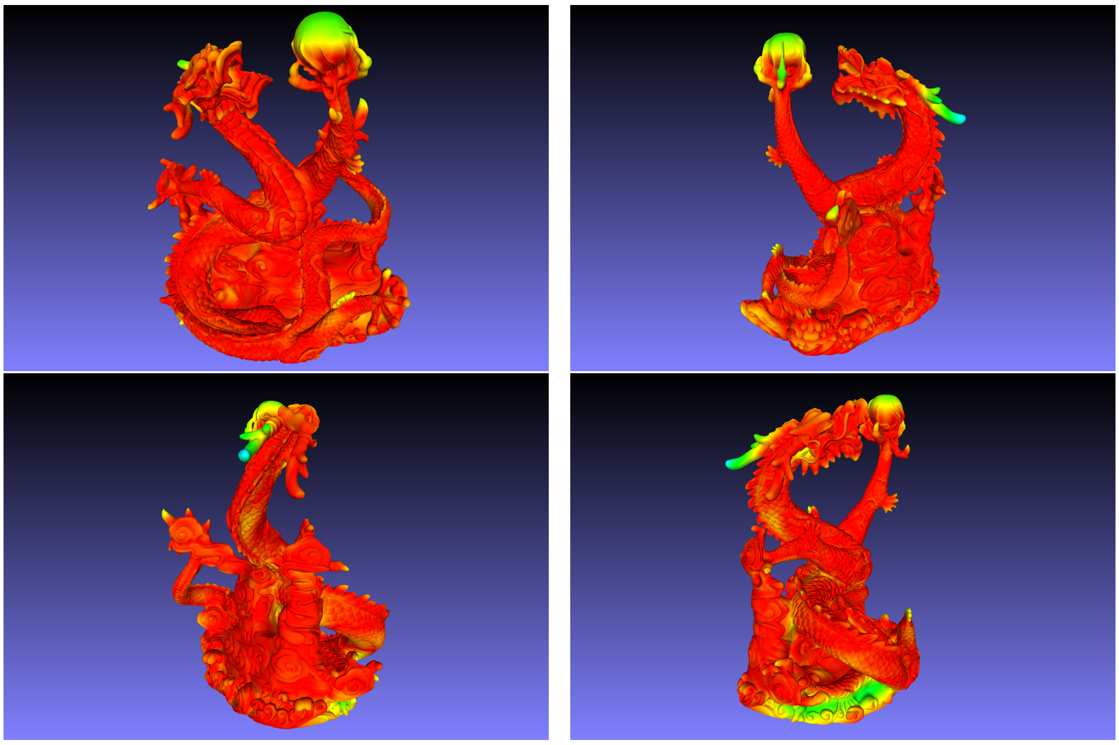

Figure 4.19: Inaccuracy on our reconstruction compared to the the ground-truth model. The colour gradient goes from red where the error is small to blue where it is high.

Figure 4.19 emphasizes even more the discussion we made before. In the regions where the error is high, our method fails to reconstruct because of the high resolution we start with. We can notice that on the body of the dragon, where we wanted to see if our method was actually able to recover the details, the error is very small. There are some parts here and there where the error is higher on the body, but we need to consider that the test was done with only 8 views of the object, so there might be hidden or too far areas from the camera to precisely get the details.

4.2 Spherical harmonics light

This section present the result of one experiment we did using the Spherical Harmonics light model. The target geometry is the Stanford bunny again, but this time the light is not modelled as in eq. (3.5) but is defined as a spherical function which is used to initialise the coefficients in our representation. Again, the resolution of the images is 256×256. The light initialisation function used is a modification of a cosine lobe. To be precise, we defined

$$f(\phi, \theta) = 5 \cdot max\left(cos\left(0.9 \cdot \theta\right), 0\right) \tag{4.1}$$

as initialisation function. This produces a cosine lobe that is a bit more smoothed out and does not go to zero at $\theta = \frac{\pi}{2}$ but a bit later.
The first test was without using any explicit initialisation, but the reconstruction was failing particularly badly for the head. Here we show the status of the reconstruction after about 170 iterations.

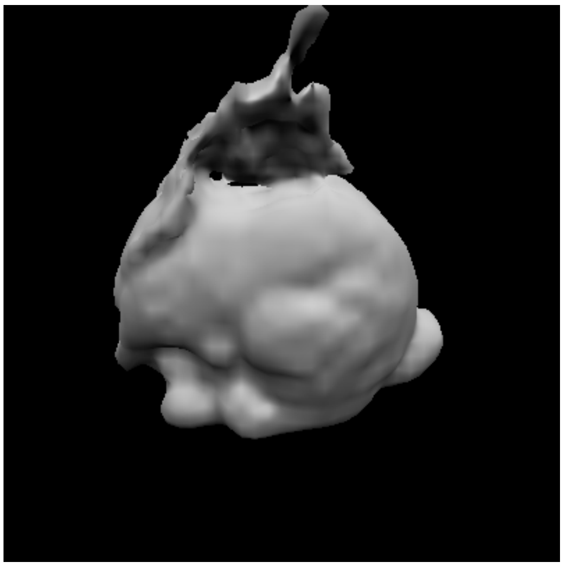

Figure 4.20: Reconstruction status starting from a simple sphere after ≈ 170 iterations. We can see how the reconstruction in the head area is not working.

The body of the bunny was being reconstructed quite well, considering we where starting from a simple sphere, but the minimization probably run into a local minima for the head region and was not able to get out from that. Later we will show the result of the reconstruction using the SH light model, but there we initialize our SDF with a simplified and smoothed version of the mesh as shown by the starting images in fig. 4.21. The fact that the reconstruction did work out in the previous experiment in section 4.1.3, where we used the direct light model, gives us the opportunity to reason why it was not the case here. Recalling eq. (3.5), we explained how the light is represented by a Dirac delta function. This means that in our rendering equation, eq. (3.6), only one direction will be selected to compute the shading of the surface. Since the light is directional, we are going to have the most information on the geometry from our image as possible. This happens because each visible point on the surface for a certain view, will be proportional to how much the normal is aligned with the light direction. However, when we use spherical harmonics the light comes from many directions. Moreover, the directions from which most of the light may be coming from, could be almost perpendicular to the normal or even occluded. This leads to images where it's much

harder to tell how the surface is oriented since multiple combinations could be possible. In fact, we could set up a light where the difference in the normals orientations are almost not visible, which would make the reconstruction extremely difficult. This clearly makes the minimization much more difficult and we get the scarce result showed in fig. 4.20.
Below the starting and target images, followed by the final rendering of our SDF after the minimisation are presented.

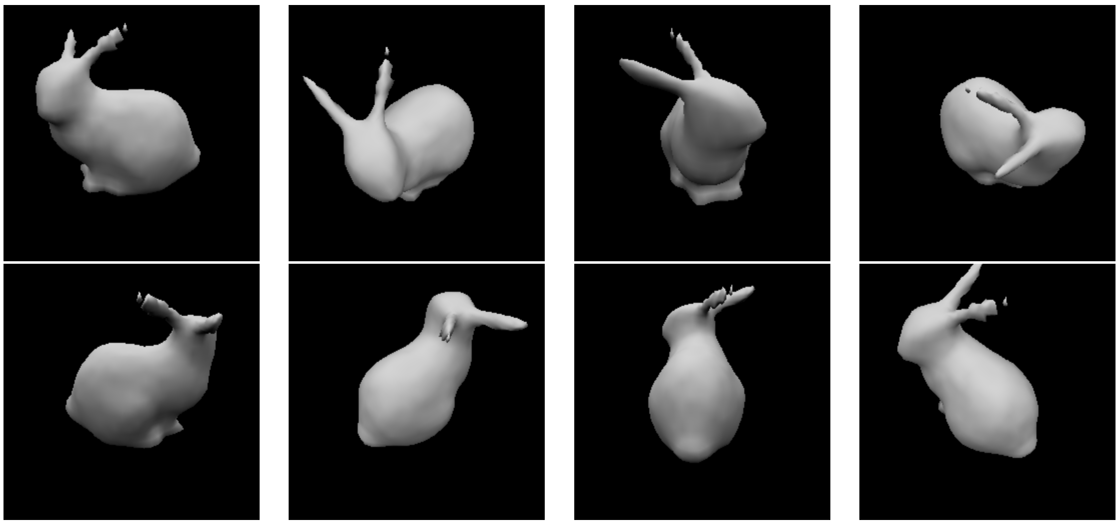
Figure 4.21: The 8 start views for the Stanford bunny test using the spherical harmonics light model. We use a simplified and smoothed version of our ground-truth model to initialise our SDF.

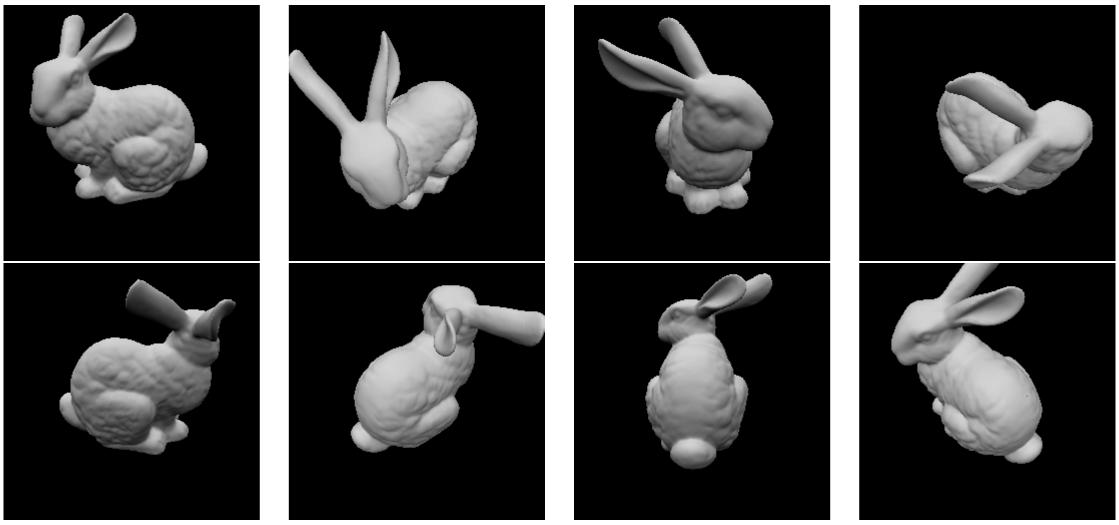
Figure 4.22: The 8 target views for the Stanford Bunny test using the spherical harmonics light model. We can notice how the illumination is smoother and more realistic compared to fig. 4.10.

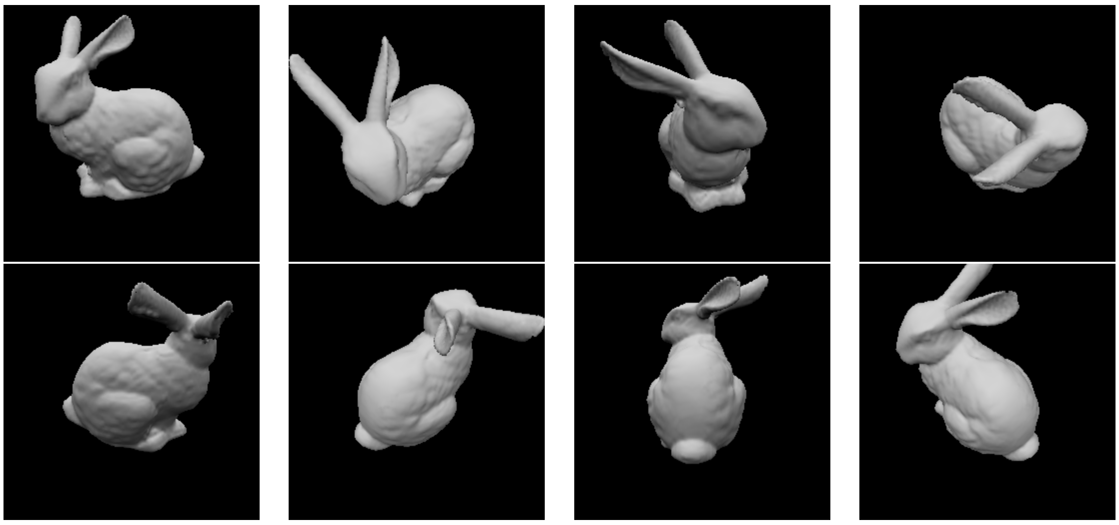

Figure 4.23: Rendering of our SDF grid at the end of the minimization process. We can acknowledge how the initialization makes the reconstruction much better compared to fig. 4.11, especially for the ears of the bunny.

We can already recognize from the images in fig. 4.23 how the reconstruction was able to reconstruct the bunny model and capture most of the details on the body. Also, this time the ears are reconstructed much better compared to the direct light example showed in section 4.1.3. The reconstruction is even able to reconstruct the interior part of each ear. This is of course because of the much better initialization we provided for this test. One thing we can notice, comparing fig. 4.22 and fig. 4.23 is that the reconstruction in the final rendering is slightly smaller compared to the target images. The graph of the energy helps us to better explain what is happening here.

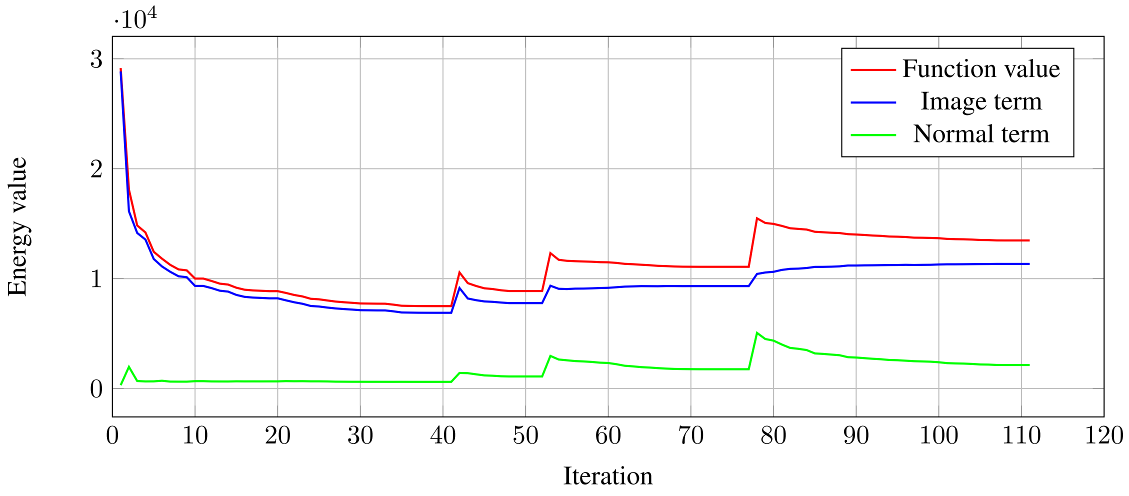

Figure 4.24: Plot of the energy value and the two single terms for the Stanford Bunny using the spherical harmonics light model.

We can see from fig. 4.24 how the behaviour of the energy for this test is similar to the one in fig. 4.12. One big difference is that the better initialization does not make the Normal term increase too much because the geometry changes are not as big. The final Image term value is also much smaller in this last test with a value of ≈ 11334 compared to the one in section 4.1.3 where the final value was ≈ 33479. This demonstrates how much our algorithm benefits from a good initialization, which is no surprise. One thing that it's interesting to see is how the energy in fig. 4.24 behaves after the first refinement step around iteration 42. In fact, we have the classical spike in the graph that shows up every time we increase the grid's resolution. From that point on, the algorithm is not able to further reduce the energy lower than what it was able to do with the first resolution. We started this test with a resolution of $41 \times 42 \times 37$ and used a resolution multiplier $s = 1.4$ which lead us to a SDF grid at the end of size $110 \times 113 \times 99$. Our interpretation of the energy behaviour is that the step used for the resolution is too high. This creates a behaviour where the new resolution is able to capture more details locally but is't too small to allow the algorithm to refine the current level of details it is able to capture. We can see how in the end, the minimization is not able to improve the Image term any more (from iteration ≈ 80) and is just readjusting the SDF grid values to match the condition explained in eq. (1.4).
The resulting images in fig. 4.23 are still visually very similar to the target one in fig. 4.22. To give a better idea of how precise the reconstruction was, we provide the output of the *Hausdorff Distance filter* from Meshlab for the initialisation mesh and our final output compared to the ground truth model.

	Minimum	Maximum	Mean
Initialisation model error	0	0.116	0.0352
Initial error w.r.t BBox diagonal	0	0.0308	0.0094
Error after our processing	0	0.087	0.0263
Error on our reconstruction w.r.t BBox diagonal	0	0.0234	0.007

Table 4.4: Hausdorff distance on the initialization used and after our processing. The bounding box size is the same as previous experiment (≈ 3.75).

As one can read from the data reported in table 4.4, we were able to reduce the maximum error of the reconstruction by 24% and the mean error on the whole model by 25%. The last results we show for this test are the images of how the error is distributed over the model and the absolute value of the difference between the starting view and the target one, and the final rendering of our SDF and the respective target.

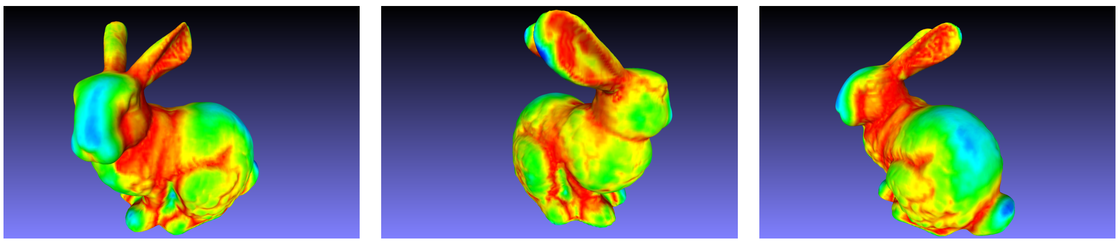

Figure 4.25: Error on our reconstruction compared to the the ground-truth model. The error increases when going from red to blue. Notice how the smaller maximum error shows much better the small variations in the reconstruction error compared to the one in section 4.1.3.

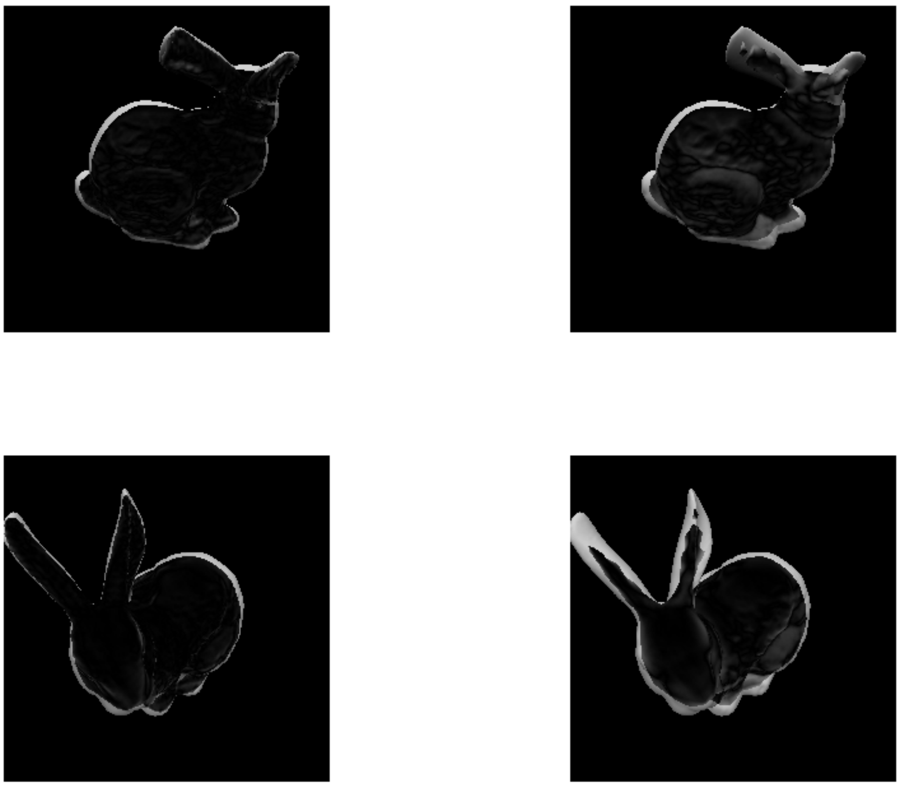
Figure 4.26: Left: difference between our final output image and the target one. Right: difference between our starting SDF rendering and target image.

Figure 4.26 clarifies even more what we discussed previously. In the left column of images we show the absolute value of the difference at the image level between our final rendering of the SDF and the target image. We can see how the details where recovered very well and it's hard to distinguish grey areas in the internal region. We can notice however how our reconstruction is smaller compared to the ground truth. This is probably due to the usage of a too high resolution for the initial SDF grid. As already explained, our algorithm does not deal well with big geometrical changes in the mesh if the size of the voxel is too small. On the right, where we compare the initialisation and the target images, we can see how there are some regions where the details are missing, especially for the top-right image. One thing that is interesting to see, in the bottom-right image, is how the top off the bunny's body after our simplification still matches in the rendering to the ground truth. This is especially because of the light model we are using, spherical harmonics, which tends to smooth out the rendering and "hide" the small details in the mesh.
A last observation can be made on fig. 4.25 where we see the ground-truth model coloured using the error we make on our reconstruction. The first thing we notice is how different it is from fig. 4.13 for how the colours changes much more on the bunny's body. This is because the maximum error we make is smaller, so the colour gradient is more evenly distributed over the error values. In fact, we can see from table 4.4 that the errors are much smaller than in table 4.2, also for the mean.

4.3 Middlebury Dino dataset

In this last section we present a failure case of our method. We try to use real images as target from the *DinoSparseRing* dataset from Steven M. Seitz et al. [19]. We select 8 views from the 16 available in the dataset and try to run our algorithm using the direct light model. Below we show 4 of the target photos used.

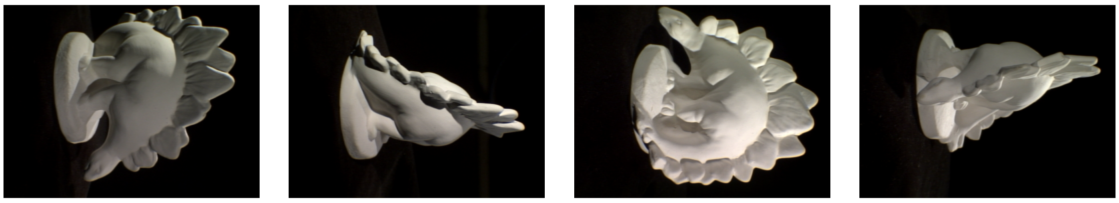

Figure 4.27: 4 of the 8 photos of the Dino model used for this test.

We initialize our SDF using the visual hull computed from the target photos. The starting rendering of our SDF can be seen below in fig. 4.28.

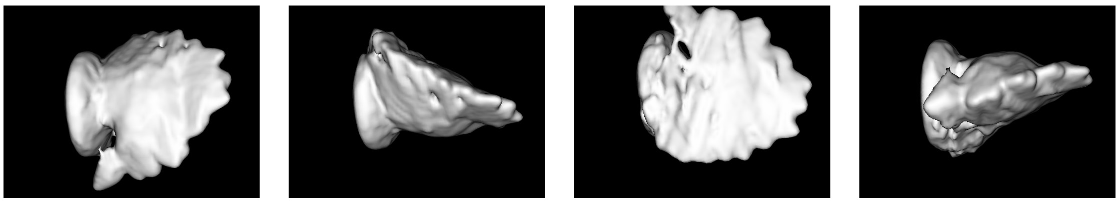

Figure 4.28: Initial rendering of our SDF.

Next we present the final rendering of our SDF when the algorithm ended.

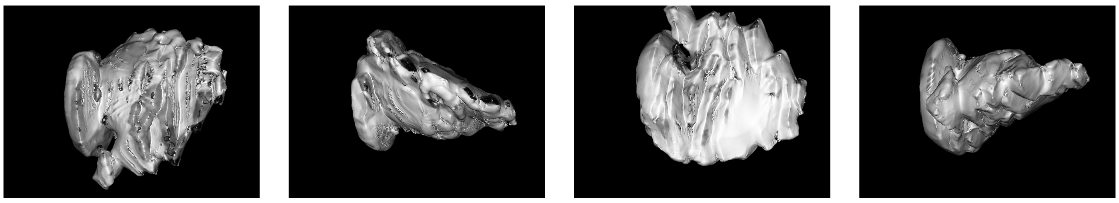

Figure 4.29: Final rendering of our SDF.

As expected, our algorithm is not able to properly match the input target images in fig. 4.27 even though the initialisation using the visual hull already provides a good starting point. In fig. 4.29 we can see the rendering of our SDF at the end of the minimization. The lack of freedom in the surface albedo and light parameters lead to a very bad output where the minimization tried to match the target images by making the output rendering darker but did not change the geometry correctly. This was expected by us but we still decided to try and see the results, in particular to show the limitations of our method through a failure case.

4.4 Runtime

As mentioned before, we implemented our code using C++. The code is still unoptimized and the performance are not optimal[1]. The time it takes to run a test strongly depends on the image size. As one can imagine, since our algorithm is based on ray tracing, increasing the image resolution affects performance significantly. The overhead due to the automatic differentiation library is also something that we must take into account. Although we only enable it when needed, it still affects performances, especially when the image resolution is high.
Another important point it's the minimization algorithm. As explained in algorithm 1, we use the backtracking algorithm to find the best step length at the current value of E along the search direction ∇E_{Ω}. The downside of backtracking is that it has to evaluate the target function at multiple steps length along the search direction and find the largest one that gives us a decrease in function value that is large enough for the parameters given. This means that our code has to render all the views of our SDF until it finds the best step-length. This can become very expensive to do, especially if the number of views is high, the resolution also and our SDF grid is very large. Table 4.5 gives an overview of the time for all the tests presented.

Target model	Image resolution	N. views	Iterations	Time (in minutes)
Cube (DL)	256×256	8	262	180
Sphere with noise (DL)	256×256	8	192	90
Stanford bunny (DL)	256×256	8	421	520
Pearl dragon (DL)	512×512	8	335	2400
Stanford bunny (SH)	256×256	8	110	360

Table 4.5: Runtime for each test. Every row reports the target model, the image resolution, the number of views used, the number of iterations and the time in minutes. (DL) direct light model, (SH) spherical harmonics light model.

The numbers in table 4.5 are of course not optimal, but as mentioned there is still a lot of room for improving the performance of our system. First thing to consider is that the whole program is single-thread. Ray tracing is an algorithm that is parallel by nature, so the time it takes to run could benefit a lot from a parallel implementation. We did not use many views in our tests (8 in all cases). Of course the time it takes to run the program would increase linearly with the number of target views in use.

[1]All tests were executed using Ubuntu 16.04 on a Intel Core I7-4710HQ@2.5GHz

5 Conclusion and future work

In this thesis we explored the possibilities that differentiable rendering offers regarding 3D Multi-view reconstruction. Of course, this is just the tip of the iceberg and much more work and research could be invested into it. We also introduced a generic framework in C++ for differentiable rendering and an implementation of SDF rendering using dense grid representation. We showed extensively in chapter 4 what our method is able to do for two different light models and using different initialization methods. In particular, we showed how we are able to recover small details on the target mesh starting from smoothed out approximations. Moreover, our energy formulation in eq. (3.11) does not make any assumption on the rendering function $f(\Theta)$. This is possible because we use automatic differentiation to compute the derivatives needed. This leaves the door open for future expansions. At the moment, we are limited by the assumptions we explained regarding our shading model in eq. (3.6). We also showed that our model is strongly affected by the grid resolution we are using, which was expected. This is a limitation that needs to be addressed in a future version of the algorithm. We also did not have time to explore in deep how the λ term on our Normal term affects the output of our reconstruction.
As already mentioned, we think that a lot of expansions of this work are possible. One of the first would be to use a better representation for the SDF grid. The dense grid we use now is very memory inefficient and quickly increases the number of variables in the minimisation. A possible improvement would be to use and Adaptive Octree structure to store our SDF. Another important aspect are performance, which at the moment are quite poor. A great first improvement would be to make the code parallel on the CPU and then extend it even further to the GPU. We already told how ray tracing is an algorithm that is parallel by nature. Another important part of the project which strongly affects the final output is the minimization algorithm. We use Backtracking along negative gradient direction for the moment, but there are others that could be able to converge faster and also use better search direction. Last thing is the multi-scale approach used, we explained how the resolution is increased using a fixed multiplier. This is very delicate and requires a lot of manual adjustments. A better way would be to implement something, based on Adaptive Octree, that refines the grid only when and where needed. This would also make our method much more robust and provide a solid base to test it on real images captured using real objects. Another possible extension to this work would be to add new shading models and make the rendering function we approximate in eq. (3.6) more realistic and general.
Overall, we believe that differentiable rendering can be very powerful in 3D Multi-view reconstruction,

especially for the level of realism it offers in the image generation. Also, signed distance function can be a very good geometry representation given the advantages they provide compared to triangle meshes like changes in resolution and topology. We hope that this work will serve as a base for more general and robust 3D reconstruction algorithm based on differentiable rendering.

Bibliography

[1] Matthew M. Loper and Michael J. Black. OpenDR: An approximate differentiable renderer. In *Computer Vision – ECCV 2014*, volume 8695 of *Lecture Notes in Computer Science*, pages 154–169. Springer International Publishing, September 2014.

[2] Steven M. Seitz, Brian Curless, James Diebel, Daniel Scharstein, and Richard Szeliski. *A comparison and evaluation of multi-view stereo reconstruction algorithms*, volume 1, pages 519–526. 2006.

[3] Y. Matsushita C. Wu, B. Wilburn and C. Theobalt. High-quality shape from multi-view stereo and shading under general illumination. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2011.

[4] F. Langguth, K. Sunkavalli, S. Hadap, and M. Goesele. Shading-aware multi-view stereo. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016.

[5] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004.

[6] O. Yilmaz and F. Karakus. Stereo and kinect fusion for continuous 3d reconstruction and visual odometry. In *2013 International Conference on Electronics, Computer and Computation (ICECCO)*, pages 115–118, Nov 2013.

[7] Sameer Agarwal, Yasutaka Furukawa, Noah Snavely, Ian Simon, Brian Curless, Steven M. Seitz, and Richard Szeliski. Building rome in a day. *Commun. ACM*, 54(10):105–112, October 2011.

[8] Stanley Osher and Ronald Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer, 2003.

[9] Michael Goesele, Noah Snavely, Brian Curless, Hugues Hoppe, and Steven M. Seitz. Multi-view stereo for community photo collections. *2007 IEEE 11th International Conference on Computer Vision*, pages 1–8, 2007.

[10] Yasutaka Furukawa and Jean Ponce. Accurate, dense, and robust multiview stereopsis. *IEEE Trans. Pattern Anal. Mach. Intell.*, 32(8):1362–1376, August 2010.

[11] B. Semerjian. A new variational framework for multiview surface reconstruction. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2014.

[12] U. Naumann P. Hovland M. Bcker, G. Corliss and Boyana Norris. Automatic differentiation: Applications, theory, and tools. In *number 50 in Lecture Notes in Computational Science and Engineering*. Springer, 2005.

[13] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson Surface Reconstruction. In Alla Sheffer and Konrad Polthier, editors, *Symposium on Geometry Processing*. The Eurographics Association, 2006.

[14] Alec Jacobson, Daniele Panozzo, et al. libigl: A simple C++ geometry processing library, 2016. http://libigl.github.io/libigl/.

[15] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *COMPUTER GRAPHICS*, 21(4):163–169, 1987.

[16] Robert Green. Spherical harmonic lighting; the gritty details. *Archives of the Game Developers, Conference*, 56:4, 2003.

[17] Mark Sussman, Peter Smereka, and Stanley Osher. A level set approach for computing solutions to incompressible two-phase flow. *J. Comput. Phys.*, 114(1):146–159, September 1994.

[18] Danping Peng, Barry Merriman, Stanley Osher, Hongkai Zhao, and Myungjoo Kang. Regular article: A pde-based fast local level set method. *J. Comput. Phys.*, 155(2):410–438, November 1999.

[19] Steven M. Seitz, Brian Curless, James Diebel, Daniel Scharstein, and Richard Szeliski. A comparison and evaluation of multi-view stereo reconstruction algorithms. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 1*, CVPR '06, pages 519–528, Washington, DC, USA, 2006. IEEE Computer Society.