FACULTY OF ENGINEERING

DEPARTMENT ELECTRICAL ENGINEERING – ESAT

KATHOLIEKE UNIVERSITEIT LEUVEN

INTEGRATION OF MULTIPLE OVERLAPPING RANGE IMAGES

Jordi Jofre Ponsatí

Universitat Politècnica de Catalunya (Barcelona)
Master of Science in Telecommunication Engineering & Management
Master of Engineering: Electrical Engineering (ICT)

Supervisor:
Geert Willems

Promotor:
Luc Van Gool

To my supervisor, Geert Willems, for all the help that he offered me every time that I needed.

To my family, who always wants the best for me, for encouraged me and allowed to live this great experience.

To my friends, for all the really nice times that we shared together in Belgium during these months.

ABSTRACT

The work described in this document continues the developing of an online 3D reconstruction pipeline for the ESAT-PSI department of K.U. Leuven (Belgium). The idea of this 3D reconstruction pipeline is that only a digital photo camera and Internet connection is necessary for a user to reconstruct scenes in 3D. The process of obtaining a 3D model from the reconstruction pipeline involves the following main three phases: the data acquisition step of the desired object, the data 3D reconstruction of the partial views and the integration of partial reconstructions in one single 3D model.

In the data acquisition phase, some regular images are taken by a consumer-grade digital camera of the target object and uploaded to the reconstruction server. In the reconstruction step, the position (and internal parameters) of the camera is computed for each image. Stereo algorithms are used to create partial reconstructions from each image that are all aligned in a common coordinate system. Both steps have already been developed. In the integration phase, we aim to all these partial reconstructions into single 3D model representation. It is this place that being developed in this Master Thesis.

A volumetric integration technique for merging multiple aligned overlapping range images based on the Marching Intersections algorithm is implemented. Furthermore, several techniques are implemented to improve the quality of the 3D final representation. These techniques are:

- A filtering procedure of the partial reconstructions.
- A weighted function according to the data input confidence.
- A triangular mesh-based hole-filling algorithm.
- An algorithm for creating a texture by stitching color information from the set of RGB input images using camera's visibility information.

We analyze and obtain conclusions about the results of the implemented integration algorithm and how works the proposed improving techniques. Finally, we analyze a comparison between our application and the volumetric integration algorithm called VripPack.

CONTENTS

1. INTRODUCTION

In order to get the complete 3D representation of an object, we must extract 3D information for large set of cameras and this often leads to a time-expensive process. Reconstructing an object's 3D shape from a set of cameras is a classic vision problem. In the last few years, it has attracted a great deal of interest, partly due to the number of application both in vision and in graphics that require detailed reconstructions like robots vision, artificial intelligence ,CAD analysis, cultural heritage, entertainment, security, etc.

Today, 3D models are used in a wide variety of fields. The medical industry uses detailed models of organs. The movie industry uses them as characters and objects for animated and real-life motion pictures. The video game industry uses them as assets for computer and video games. The science sector uses them as highly detailed models of chemical compounds. The architecture industry uses them to demonstrate proposed buildings and landscapes through Software Architectural Models. The engineering community uses them as designs of new devices, vehicles and structures as well as a host of other uses. In recent decades the earth science community has started to construct 3D geological models as a standard practice.

Three-dimensional (3D) image acquisition systems are rapidly becoming more affordable, especially systems based on commodity electronic cameras. At the same time, personal computers with graphics hardware capable of displaying complex 3D models are also becoming inexpensive enough to be available to a large population.

As a result, there is potentially an opportunity to consider new virtual reality applications as diverse as cultural heritage and retail sales that will allow people to view realistic 3D objects on home computers. These trends, coupled with Internet increasing bandwidth, are making use of 3D models beyond the well established games market to a new applications ranging from virtual museums to e-commerce.

The work described in this document continues the developing of an online 3D reconstruction pipeline for the ESAT-PSI department of K.U. Leuven (Belgium). The idea of this 3D reconstruction pipeline is that only a digital photo camera and Internet connection is necessary for a user to reconstruct scenes in 3D.

In order for such systems to be practical use, the 3D data extraction process is expected to be fast and reliable. The process of obtaining a full 3D model reconstruction from the reconstruction pipeline involves the following main three phases:

- Data Acquisition of the desired object.
- Data Reconstruction of the partial views.
- Integration of partial reconstructions in one single 3D model.

The data acquisition phase corresponds to the input data of the 3D reconstruction system. These input data are regular 2D images taken by a consumer-grade digital camera of to the target object that we desire a 3D model. A client-side application tool for uploading images to the reconstruction server, called the ARC Server [1] (now on we will refer to it as the ARC Server) is available.

After uploading the images to the ARC Server, the reconstruction process starts. When it finish, we obtain from the ARC Server, for every uploaded image: the camera calibration parameters, the depth map, the quality count map and the texture file related to this image. Using these data, we can obtain a 3D reconstruction for every uploaded image.

One single image cannot cover the full object's surface. Several images are taken from different points of view to cover all object's' surface. The last step of the reconstruction pipeline is the integration of the partial reconstructions in one single 3D model. The aim of this Master Thesis is use all the data provided from the ARC Server to implement a fast and efficient volumetric algorithm for the integration of multiple aligned overlapping images that does not consume too much memory. All properties are needed if we want to be able to run it on clients' computers.

Furthermore, other procedures to improve the final 3D model are implemented like a filtering step based in diffusion / erosion and blob detection of the input data, a mesh based hole-filling algorithm and color detail stitching texture information.

2. DATA ACQUISITION

In the data acquisition phase, we collect the data of the target object to be reconstructed. The input data in this case is set of 2D images taken with a digital camera from several points of view of the object.

Our acquisition device is a consumer-grade digital camera. The type of camera used will clearly influence the outcome. A higher image resolution, will of course, result in longer upload and processing times and more detailed depth maps at the end. More light sensitive cameras will also improve the results. The models used in this work were taken with a Canon Digital IXUS 430 of 4 Megapixels.

Our data acquisition procedure involves recording a sequence of images of an object from several points of view. We need to take into account that, these images will be uploaded to The ARC Server [1], therefore, this sequence of images must be taken in such a way that facilitates the reconstruction and avoid the frustration of obtaining fail results.

No prior knowledge on camera calibration is known, so all information must be recovered from the images It is therefore very important that enough information is present in the images, like:

-General motion of the camera

-General structure in the scene

-Enough overlap (only points that are visible in at least 3 images are useful)

In a first step, the camera calibration is computed on the ARC Server [1] by matching images. As a result, texture information (color, intensity) of the scene/object is critical, enough texture must be available on the object and the appearance (light conditions, not moving scenarios) of object must remain constant.

As we just said, lighting conditions are also important, we need the most uniform and stable conditions as possible, then, we can find the best lighting conditions in overcast sky and the worse ones in days with changing conditions.

In the Figure 1, we can see one the acquisition motion example, the gray cones represents the shoot positions of the camera and the white dotes, the matches between images found by the ARC Server [1] after examining the uploaded images.

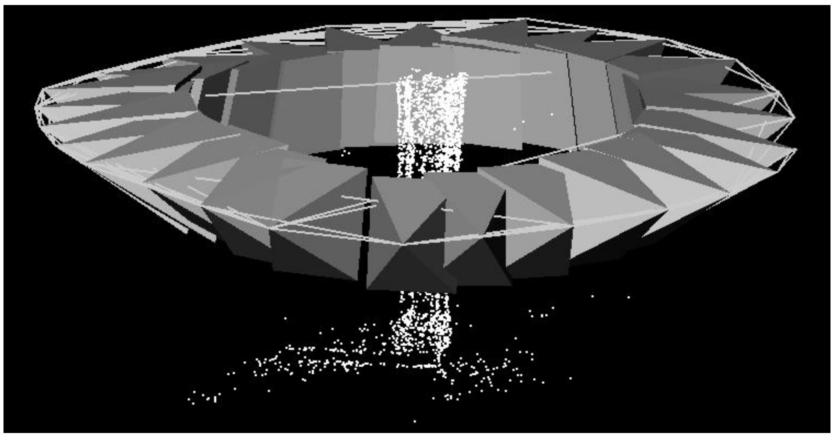
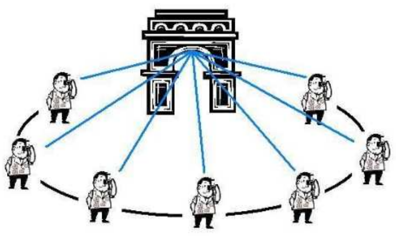


Figure 1: Acquisition motion of one object.

In order for the ARC Server [1] to obtain a successful 3D reconstruction, we need to take in account some tips before taking the sequence of images. The most suitable way of obtaining the set of images is:

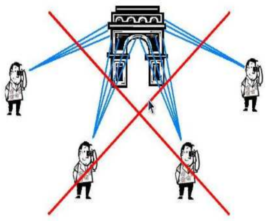


-Shoot a picture of the same location for every step made in the shooting sequence. This results in multiple pictures of the same scene, but viewed from slightly different sides.

-Walk with the camera in an arc around the scene, while keeping the scene in frame at all times.

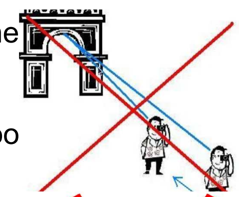
-A minimum of five or six images are required for a good reconstruction and each area of the object must be seen at least for three cameras.

In the other hand, we should avoid the following:



-Do not pan from the same location, as if you were recording a panorama.

-Do not shoot multiple panoramas from different viewpoints.



-Do not walk in a straight line towards or inside the scene you want to reconstruct.

-The viewing angle between images should not be too small, due to adjacent images should not be too far apart.

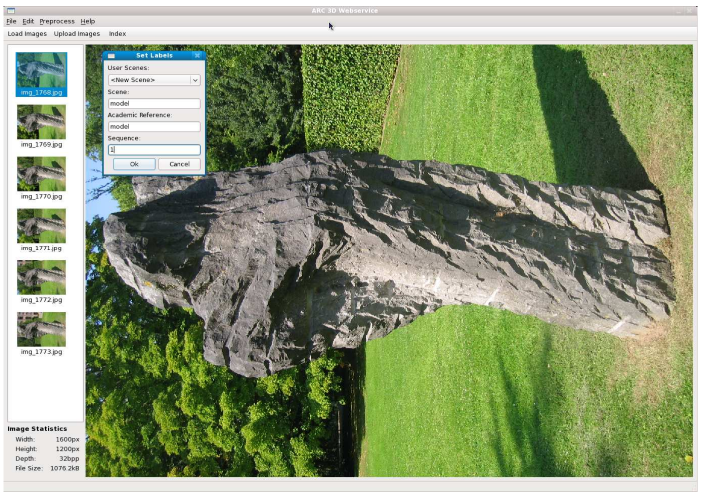
-Do not take pictures of a planar scene. These scenes do not contain enough information for camera calibration.

-Do not use a turntable with a static camera and a scene that rotates on a perfect circle in front of the camera.

-Avoid blurry images, cause problems in the reconstruction step.

-Objects blocking the scene, like trees, make the reconstruction of hidden parts impossible.

Finally, the acquisition step is over when we upload the set of images to the ARC Server [1]. For this, we have available the ARC Upload tool where we can sub sample the images to reduce the upload time and a filter to detect the too blurry images. In Figure 2 we have a screen-shoot of ARC Upload tool.



User Scenes: <New Scene>
Scene: model
Academic Reference: model
Sequence: 1

Image Statistics
Width: 1600px
Height: 1200px
Depth: 32bpp
File Size: 1076.2kB

Figure 2: ARC Upload Tool.

3. DATA RECONSTRUCTION

The 3D web service is meant to create 3D reconstruction from a wide variety of images. Our input data is just several photos took for one user for an arbitrary object without prior information about camera parameters (position, radial distortion, etc.). Because no user interaction is possible once the images have been uploaded, an important prerequisite is the need for robustness on the part of the ARC Server. The actual 3D reconstruction pipeline consists of roughly for steps:

First, a step that computes a set of image pairs that can be used for matching. In this first step, the images are subsampled. Since images can be uploaded in non-sequential order, we figure out which images can be matched. This is task of the Global Image Comparison algorithm with yields a set of image pairs that are candidates for pairwise matching.

Next, a step that performs the Pairwise and Projective Triplet Matching and the Self-Calibration is computed. All possible matching candidates of first step are now tried. Based on the resulting pairwise matches, all image triplets are selected that stand a chance for projective triplet reconstruction. The results are fed to the self-calibration routine which finds the intrinsic parameters of the camera.

After that, a step that computes the Euclidean reconstruction and upscales the result into full resolution is computed. In this step, all image triplets and matches are combined into one 3D Euclidean reconstruction.

Finally, a step that is responsible for the dense matching, yielding dense depth maps for every image. The ARC Server [1] will send an email notification when the reconstruction step finishes with a link where we can download a .zip file where we can find the following files:

- Camera calibration files
- Dense depth map files
- Quality count map files
- Texture files

The results can be inspected by means of model-viewer tool. Given a selected image, a triangulated 3D model can be created for the current image, using the depth map and camera this image. Every pixel of the depth map can be put in 3D and a triangulation in the image creates a 3D mesh, using the texture of the current image. Every depth map has its corresponding quality map, indicating the quality or certainty of the 3D coordinates of every pixel. The quality and accuracy of the resulting depth maps is proportional to the size of the input images.

At this point, the user has partial reconstructions available for each of the uploaded images. In order to create a full 3D model, these partial reconstructions need to be loaded into 3D packages, like MeshLab.

Here is where we want to extend the actual K.U. Leuven pipeline. We propose an automatic integration algorithm for merging all the partial reconstructions in one single 3D model. We need a fast and memory efficient algorithm to be able to run on clients' computers.

We will use the files obtained from the ARC Server reconstruction in our integration algorithm. For this reason, we detail the contents of each one in the following sections.

3.1. Camera Files

The camera files we record the camera calibration parameters of our set of images. Camera calibration is the process of determining internal or intrinsic and external or extrinsic camera parameters.

The intrinsic camera parameters fully describe how the three dimensional world is projected onto a two dimensional plane. These parameters correspond to camera geometry and optics.

The extrinsic camera parameters describe the pose of the camera. The pose has two components, namely the position of the camera and its orientation. The position description is described by a translation vector ***t*** and the orientation with a rotation matrix ***R***.

The camera model that will be used throughout this text is the linear pinhole model. This camera model is inspired in the camera obscura: a black box, one side of which is punctured to yield a small hole and with photosensitive paper on the opposite side. The pinhole camera model describes the mathematical relationship between the coordinates of a 3D point and its projection onto the image, where the camera aperture is described as a point and no lenses are used to focus light.

The mathematical set-up of the pinhole model is shown in Figure 3. It shows a 3D point ***M***, projected in a camera with center ***C***.

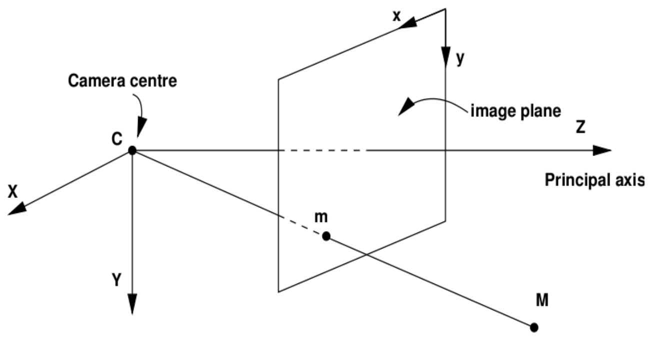


Figure 3: Pinhole Camera Model.

The projection ***m*** of ***M*** in the image is the intersection of the image plane with the ray formed by connecting ***C*** with ***M***. The local coordinate system of the camera is located in the world using translation ***t*** and rotation ***R***. The equation for this projection is:

$$m \approx K(R^T | - R^T t)M \tag{3.1}$$

$$\mathbf{K} = \begin{pmatrix} \alpha_x & s & u_x \\ 0 & \alpha_y & u_y \\ 0 & 0 & 1 \end{pmatrix} \tag{3.2}$$

Where ***K***, is the matrix holding the internal camera parameters. These are:

α_x, α_y = The focal length in pixels *x* and *y* in direction α_x and α_y. The focal length of the lens is split into a horizontal and a vertical component.

u_x, u_y = Coordinates of the main point being the intersection of the optical axis with the image plane.

s = The skew *s*. It describes the extent to which the pixels are distorted compared with a rectangle usually this value is equal to zero since CCD sensor and the pixels are king right thoe.

The ***K***-matrix takes into account all linear conditions. It does not take into account non-linear phenomena, like radial distortion.

The model is a simplification of reality, but sufficient in practice. The model does not include, as we said, geometric distortions or blurring of unfocused objects caused by lenses and finite sized apertures. It also does not take into account that most practical cameras have only discrete image coordinates. This means that the pinhole camera model can only be used as a first order approximation of the mapping from a 3D scene to a 2D image. Its validity depends on the quality of the camera and, in general, decreases from the center of the image to the edges as lens distortion effects increase.

Some of the effects that the pinhole camera model does not take into account can be compensated for, for example, by applying suitable coordinate transformations on the image coordinates. Others effects are sufficiently small to be neglected if a high quality camera is used.

Here we have one example of camera file. We can find all the parameters that we talk about previously:

```
1937.83 0 799.725
0 1941.48 612.497              calibration matrix
0 0 1

-0.127652 0.111366 0           radial distortion

0.877744 0.0818358 0.472089
0.0277686 -0.992338 0.120391   rotation matrix
0.478324 -0.0925629 -0.873292

-8.79053 -2.27521 33.4724      translation matrix
1600 1200                      image resolution
```

3.2. Depth map files

The depth map can be seen as a single channel containing float values, namely, the distance from 3D point, projected onto respective pixel, to the camera center. A depth zero value, which would mean the 3D point lies on the camera center, and thus is impossible. A depth zero value is used to denote invalid depth values.

Given a series of two-dimensional images taken from different points of view, it is possible to extract a significant amount of extra information about the scene being captured. One of the most useful of these pieces of information is knowledge about the relative depth of objects in the scene.

Note that, the information provided by only two images may be insufficient to determine reliable correspondences. It is possible to reduce the ambiguity by extending point correspondence to multiple images. That is why one of the acquisition requirements is that each object area must be seen at least for three images. ARC Server performs the dense matching, yielding dense depth maps for every image.

In order to visualize the depth maps, we can scale the depth values into an 8 bits grey-scale image. Each pixel of this grey scaled image, stores the corresponding depth reading (*3D data*) of the object in 3D world. According to this scaling, an almost black pixel represents the closest one to the camera shoot position (black means invalid depth value), and a white pixel represents the farthest.

In the Figure 4 on the left, we can see the original 2D image shoot by a digital camera and on the right the corresponding scaled in a grey-scale depth map. As we can check, the blackest areas, effectively, are the ones closer to the camera position, and the more light grey, the farthest.

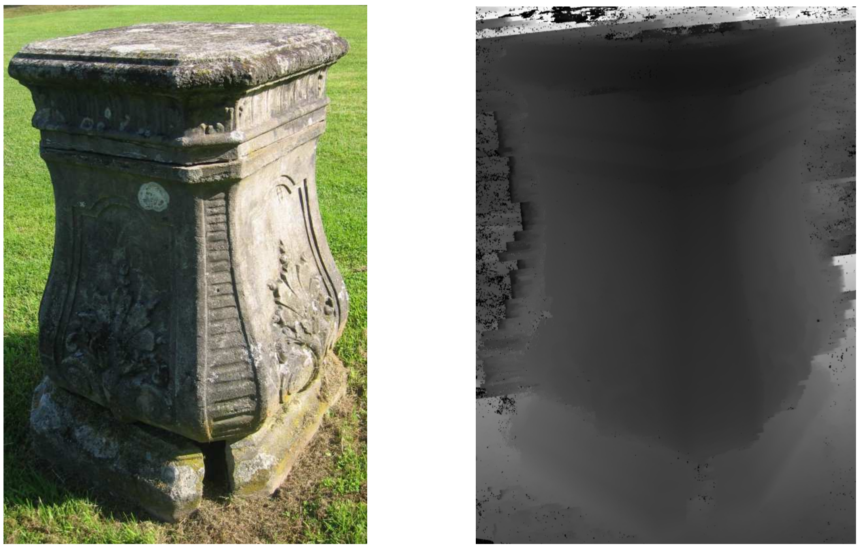


Figure 4: Left: Original user photography. **Right:** Corresponding scaled depth map representation.

3.3. Quality Count map files

The quality count map can be seen as a single channel containing integer values, namely, the number of cameras that is seen that pixel. Quality maps indicate the confidences we have in the 3D reconstruction of every pixel are part of the output. The more consistently, the more confident we are of its 3D position, hence the higher the quality measure for this pixel.

One of the requirements of our system is that every part of the object must be seen, for at least, three images, so it is necessary overlapping between images. Quality count maps indicates for every pixel of the image, for how many cameras this pixel is seen.

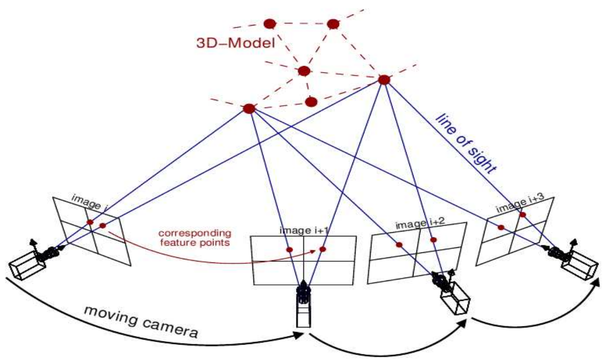


Figure 5: Point correspondences in multiple images. Count value.

In order to visualize the quality count maps, we can scale the count values into an 8 bits grey-scale image. Each pixel of this grey scaled image, stores the corresponding count reading (*number of cameras*). According to this scaling, a black represents that this pixel is not seen by any camera. As more close is a pixel to a white color, means that is seen by more cameras.

In Figure 6 on the left, a sample image took by a digital camera and on the right his count map representation. We can notice that, the flank of stone is seen by more cameras. In the other hand, the top of the stone is the area seen by fewer cameras.

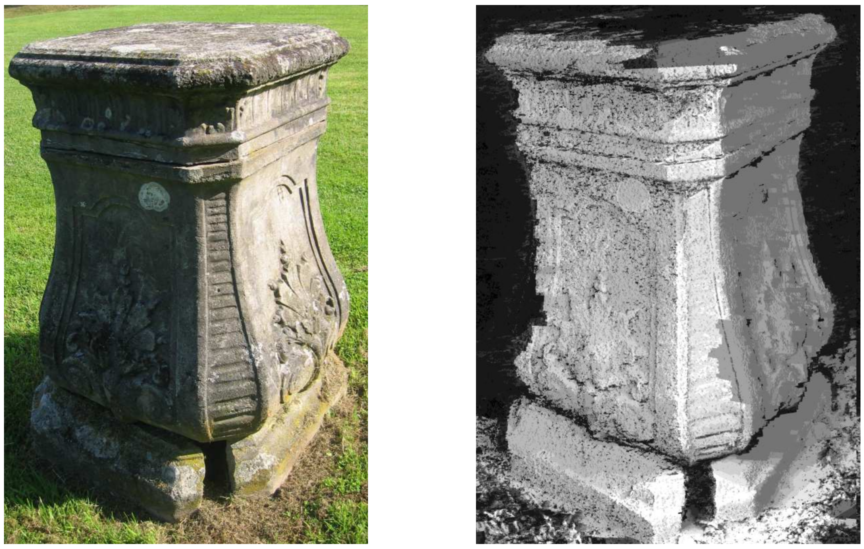


Figure 6: Left: Original sample image**. Right:** Corresponding scaled count map representation.

3.4. Input File Format

We can reconstruct 3D information of every image from the corresponding camera parameters and depth map. But using the 3D information in this format is impractical and cumbersome. The Model-viewer tool of the reconstruction pipeline can export the model in different 3D formats. In our application, we choose the PLY format due to be simple and flexible.

PLY [30] is a computer file format known as the Polygon File Format or the Stanford Triangle Format. The format was principally designed to store three dimensional data from 3D scanners. PLY files are organized as a header, that specifies the elements of a mesh and their types, followed by the list of elements itself, usually vertices and faces.

As we said, the PLY format describes an object as a collection of vertices, faces and other elements, along with properties such as color, normal direction, texture coordinates, transparency, range data confidence, and others that can be attached to these elements. A simple example of PLY can be:

```
ply
format ascii 1.0           { ascii/binary, format version number }
comment this file is a cube{ comments keyword specified, like all lines }
element vertex 8           { define "vertex" element, 8 of them in file }
property float x           { vertex contains float "x" coordinate }
property float y           { y coordinate is also a vertex property }
property float z           { z coordinate, too }
element face 6             { there are 6 "face" elements in the file }
property list uchar int vertex_index { "vertex_indices" is a list of ints }
end_header                 { delimits the end of the header }
0 0 0                      { start of vertex list }
0 0 1
0 1 1
0 1 0
1 0 0
1 0 1
1 1 1
1 1 0
4 0 1 2 3                  { start of face list }
4 7 6 5 4
4 0 4 5 1
4 1 5 6 2
4 2 6 7 3
4 3 7 4 0
```

In addition, the PLY format allows to define own elements as well. The format for defining a new element is exactly the same as for vertices, faces and edges. As we will see later, we will use this feature to define additional properties.

4. DATA INTEGRATION

This chapter is organized in the following way; first, we define some concepts related with the surface representation of the integrated 3D model. Next we give a brief summary about the literature related integration algorithms. We continue with a description of the integration algorithm implemented. Finally, some techniques for improving the 3D model are proposed and described in more detail.

4.1. Related concepts

The aim of this work is to reconstruct a 3D model from a set of 2D images, but, until now, we did not talk about how we represent and store this model, and in which way we visualize the 3D model on a computer screen. We will represent the model using triangular meshes. In the following lines we define the concepts related to triangular meshes.

A vertex or a point is a position in 3D world along with other information such as color, normal vector and texture coordinates. An edge is a connection between two vertices. A face is a closed set of edges, in which a *triangle face* has three edges. A polygon is a set of faces.

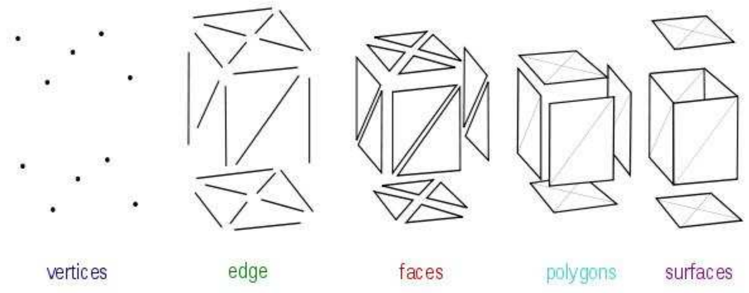


Figure 7: Mesh related concepts.

A polygon mesh is a collection of vertices, edges and faces that defines the shape of an object. The faces usually consist of triangles that simplifies rendering. A triangle mesh is a type of polygon mesh. It comprises a set of triangles that are connected by their common edges or corners. A triangular mesh is represented by two separate arrays, one array holding the vertices, and another holding sets of three indexes into that array which define a triangle. Note that, the triangular mesh definition matches with PLY format structure. The Figure 8 shows an example of triangular mesh.

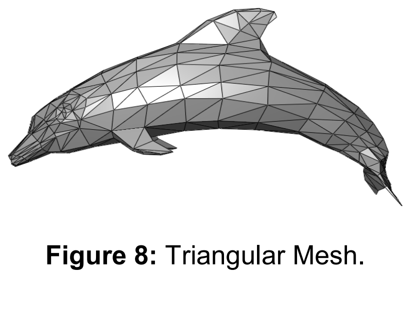


Figure 8: Triangular Mesh.

Now that we have defined these concepts, we continue with a summary of other integration algorithms in the literature.

4.2. Related work about integration algorithms

Surface integration from dense range data has been an active area of research several years. The strategies have proceeded along two basic directions: integration from unorganized points and integration from range images. These two strategies can be further subdivided according to whether they operate by reconstructing parametric surfaces or by reconstructing an implicit function.

4.2.1. Integration algorithms from unorganized points

Integration from unorganized points presumes that one has a procedure that creates a polygonal surface from an arbitrary collection of 3D points in space, in other words they do not any prior assumption about connectivity of points. In the absence of depth maps or contours to provide connectivity cues, these algorithms are the only recourse on unordered point clouds as input. Integration in this case is complete by collecting together all the points from multiple views and presenting them to the polygonal integration procedure, so it is an important constraint to have uniform data distribution of the point cloud.

Among the parametric surface approaches, also know like polygonal or mesh integration methods, Boissanat [2] describes a method for Delaunay triangulation of a set of points in 3D space. Edelsbrunner and Mücke [5] generalize the notion of a convex hull to create surfaces called alpha-shapes.

The implicit functions, also known as volumetric integration, are characterized by a volume space, defined inside, a grid, and it is ideally superimposed on the 3D scene. Examples of implicit surface reconstruction include the method of Hoppe [7] that starts from an unorganized set of points, define a signed distance function from the point set, and extracts a mesh approximating the zero set of that function. More recently, Bajaj [8] used alpha-shapes to construct a signed distance function to which they fit implicit polynomials.

The integration from unstructured data are widely applicable, but they discard useful information such as surface normal and reliability estimates, and generally belongs high computational costs and the necessity of post processing smoothing to solve possible imperfect regions. Another limitation is that a uniform distribution of the point cloud is preferred.

As a result, these algorithms are well-behaved in smooth regions of surfaces, but they are not always robust in regions of high curvature and in the presence of systematic range distortion and outliers.

4.2.2. Integration algorithms from range images

Integration from range images make use of information about how each point was obtained and the existence of connectivity information and reliability of measurements.

Among integration algorithms from range images, several parametric approaches have been proposed. Soucy and Laurendeau [9] describe a method using Venn diagrams to identify overlapping data regions. For each disjoint part, a virtual viewpoint is defined and a non-redundant integrated triangulation is obtained. Finally, all these local triangulations are connected to yield a global integrated one.

Turk & Levoy [10] employ an incremental algorithm that updates a reconstruction by eroding redundant geometry. He continues by zippering along the remaining boundaries. Last, a “consensus” step that reintroduces the original geometry to establish final vertex positions.

Pito [11] identifies for each couple of input meshes, all the triangles that sample the same surface patch. The algorithm keeps only one of them, namely the one acquired with the highest confidence. The redundant triangles are removed, and neighborhood relationships are established between the remaining patches.

Ratishauser [6] integrate triangular meshes by completely re-triangulating overlapping areas, since the triangulation is in 3D. Special care must be taken to ensure the mesh does not fold over itself.

Several algorithms have been proposed for integrating from range images to generate implicit functions. In these techniques, a volume space is defined, a grid, and it is ideally superimposed on the 3D scene and a scalar value is associated with each node of the grid, generally, a signed distance between the node and the 3D surface. Finally, a surface fitting algorithm is used for the reconstruction the surface.

Pulli [12] builds an octree from a sequence of range maps. The algorithm processes a cubical volume surrounding the object, where it checks for each cube if it’s external, internal or in the boundary of the object. The final reconstruction is obtained by generating triangles between the external nodes and those on the boundary. The output surface is also smoothed.

Conolly [13], creates an octree representation by organizing the range views into quadtrees and projecting them to the octree, marking the free space along the way.

Hilton [14], generate a volumetric implicit-surface representation from a number of range images using a local search to find the surface points from each view which are closest to a given voxel center. Heuristics are then used to determine which of these closest points to use in estimating the signed distance. The main problem with their method is that their rules for determining which points to use in the average do not guarantee that the points are all part of the same portion of the objects surface.

Wheeler [15] uses octrees to represent volumetric surface and estimates the signed distance to the object surface by finding a consensus of locally coherent observations of the surface. In this way, he eliminates many of the troublesome effects of noise in the input data.

Curless and Levoy [3], define a volumetric function based on the signed distance transform along the line of sight to the sensor. Second is the weight function, a kind of confidence value that is used when combining the implicits functions generated by different views. The surface is located where the implicit function is zero. Data from multiple views are combined by a weighted average to create an overall implicit function and weighting function. All the voxels are initialized as unseen voxels. The scan conversion of the range maps writes the visible voxel. All the voxels lying between a visible voxel and the sensor are marked as empty voxels. An unseen voxel lying between visible voxel are the ones that will be filled by the algorithm.

As a conclusion, integration from range images using parametric approaches typically performs better than unorganized point algorithms, but they still fail in areas of high curvature. Also, turns out to be easily feasible also in the cases of non uniform data distribution.

In the other hand, integration from range images using a volume fusion approach present a high computational cost, mainly due to the initialization of the distance volume. Volumetric algorithms are not as precise as mesh integration algorithms because they represent an approximation of the integrated meshes, but there are definitely more robust.

4.2.3. Our proposed method

The integration algorithms from unorganized points discard useful information such as surface normal and reliability estimates, and generally belongs high computational costs and the necessity of post processing smoothing to solve possible imperfect regions. They are not always robust in regions of high curvature and in the presence of systematic range distortion and outliers.

The reasons to discard implementing an integration algorithm from unorganized points are: First, our input data are partial 3D reconstructions. We have available for each partial reconstruction, connectivity and reliability information from the depth maps, count maps and camera parameters. Second, the high computational cost and the no robustness in high curvature regions.

Integration algorithm from range images based on parametric approaches typically perform better than unorganized point algorithms, but they still fail in areas of high curvature. For this reason, we discard to implement one integration algorithm based on parametric approaches.

For volumetric approaches, the main problem for us is the high computational cost as well as the fact that they require a large amount of memory.

Searching through the literature, we came across a volumetric algorithm by Rocchini [4], called Marching Intersections, which has the advantage over other volumetric algorithms in that it does not require a large amount of memory. Furthermore, it works efficiently and can produce high quality surfaces.

For these reasons, we start our integration algorithm with an implementation based on the one designed by Rocchini [4]. After evaluation the initial results, we propose several adaptations for our specific case.

4.3. Marching Intersections algorithm

Marching Intersections (MI) is a volumetric algorithm for the simultaneous fusion of multiple, partially overlapping range images into a triangular mesh.

The input data of Marching Intersections algorithm are triangular meshes previously registered and defined in a single Euclidean Space. As is typical for a volumetric algorithm, a reference grid is defined and superimposed on the 3D model. The grid resolution is chosen by the user and generally represents a compromise between memory needs and execution times with respect to output precision and quality. The grid resolution is defined by indicating the number of desired lines in each axis.

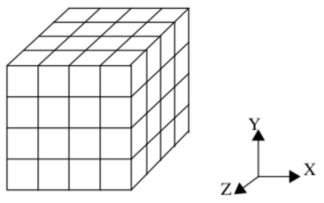


Figure 9: Reference Grid of size 5x5x5.

The main steps involved in the algorithm are: Locate the intersections between input meshes and the reference grid. The second step is merging all nearly coincident and redundant intersections belonging to different input meshes, according to a proximity criterion. This criterion depends on the grid resolution and the quality associated with the different parts or observations of the input meshes. The last step is reconstructing the merged surface from the filtered intersection set by means of the Marching Intersections algorithm.

The final integrated model representation is mainly influenced by two kinds of errors: the acquisition error and the registration error. The first one is caused by the limited accuracy of the hardware, in this case, our CCD camera. Registration error is the error due to the not perfect alignment between range images, this error is quite more critical in our integration process.

In the following sections, we detail the steps involved in Marching Intersections algorithm.

4.3.1. Scaling transformation

Our application receives as input a sequence of partially overlapping and aligned range images in PLY [30] format. Given a grid resolution chose by the user, the first step is computing the integrated bounding box of the object. A bounding box is an imaginary box where the object can fit inside. In order to calculate the integrated bounding box of the whole object, we need to calculate the bounding box of every range image and the integrated bounding box will be the higher one. In this way we can assure that the whole objects fits in the final integrated bounding box.

In order to achieve the maximal possible resolution with the given reference grid, we must apply a scale transformation for every vertex of the PLY file to fit in an efficient way the integrated bounding box in our reference grid. The following matrix is the scaling matrix, with s_x, s_y, s_z the scaling transformation vector and *x, y, z* the vertex coordinates in the space.

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \cdot S_x \\ y \cdot S_y \\ z \cdot S_z \\ 1 \end{pmatrix} \tag{4.1}$$

To compute the scale transformation we apply the next formula in the three coordinate axes *x, y, and z*:

$$S = \frac{numberGridLines}{maxIntegratedBBox - minIntegratedBBox} \tag{4.2}$$

To keep the original shape and avoid undesired deformations, we should apply the resulting lowest axis scale transformation. We need to apply the same scale transformation in every axis to keep the object proportionality and avoid undesired object deformations. We choose the lowest one to assure that the object fits inside the grid.

4.3.2. Range map discretization

The discretization step consists in detecting all the intersections between the input meshes and the reference grid. To simplify the computations, all the geometric coordinates of the input data are immersed in the space of the selected grid. The goal of the previous scaling transformation is to have all the grid lines lying on integer values. This improves the efficiency of the computations and the management of the intersections between the input meshes and grid edges.

The discretization of the input meshes occurs on a per-face basis. The procedure is the following: First, we project each face onto the orthogonal planes *XY, XZ* and *ZY*. For each plane, we calculate the bounding box of triangle. Next, for each intersection of the grid lines inside the bounding box, we check if the point is inside or outside the triangle. To determine if one point is inside or outside a triangle we compute the barycentric coordinates of the triangle.

If the point is inside the triangle, it means that the face intersect the reference grid in a point (*x,y,z*), where *x*, *y* are known because correspond to the grid line values, and *z* value is determined using the barycentric coordinates. The Figure 10 corresponds to the *XY* projection of one face. The blue dots are the ones of the defined bounding box that this face has an intersection with the grid lines in *XY* projection. Notice that effectively, *x, y* values lie always in an integer values.

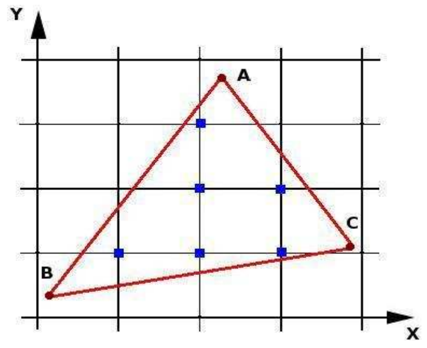


Figure 10: Discretization step in 3D.

This process leads to the construction of the 2D dynamic data structures *XY, XZ* and *ZY* containing the intersections between all the input faces and the corresponding grid lines. Each one of these 2D dynamic data structures stores the detected intersections with the grid lines orthogonal to each orthogonal plane. They are implemented as an array of arrays of intersections with size according to the number of grid lines:

```
vector< vector<Intersection> > xy;
vector< vector<Intersection> > xz;
vector< vector<Intersection> > zy;
```

For example, the entry (1,3) of the 2D pointer data structure *XY*, refers to the list of intersections between the input range surfaces and the grid which intersection points are (1,3,*z*). A data structure is defined in our implementation to record the following information for each intersection:

```
struct Intersection {
        float intercept;
        int meshName;
        bool sign;
        float direction;
        float weight;
        int meshPosition;
};
```

where:

intercept(ic):Is the numeric value (intercept) of the intersection. For example, in the XY data structure, *ic* represents the *z*-axis component of the intersection.

meshName(nm):Each bit represents an input mesh or meshes the observation belongs to. The size of an integer limits the maximal number of mesh that we can integrated, we consider that 64 meshes are enough for our purposes.

sign(sgn):Correspond to sign of the intersection. We define the sign according to the fact whether we are entering or leaving the object. If we are walking along the grid size, considering a right handle coordinate system, a `+` sign means that we are entering and a `-` that we are leaving the object.

direction(dr):Is the direction field. This field holds the component along the grid line the intersection belongs, according with the *sign* field.

weight(w):Is a value that associates an intersection with the quality of the surface sample. The value is obtained by multiplying the quality of the point (quality map) and the viewing angle from the camera. If the intersection belongs to a boundary, then the weight is equal to zero.

meshPosition(mp):Is one auxiliary field used for writing the integrated mesh result. It indicates the position of vertices in the vertices array of the mesh.

At the end of the process, each list is sorted with respect to the intersection value. Sorting ensures fast detection of nearby intersections and fast search for specific intervals.

After repeating the discretization process for all faces of each input mesh, all the intersection between the grid and the surface are determined and entered in the 2D dynamic data structures.

We highlight the compactness of the algorithm. Instead of storing the full 3D grid with cells, we just store the *XY, XZ* and *ZY* intersection list and neither, does not store the input meshes or connectivity information. The space complexity in our case is directly dependent on the surface area of the meshes, instead of being proportional to the meshes volume. We can now start with the actual merging.

4.3.3. Merger process

At this point, the 2D dynamic data structures are filled and their intersections sorted with respect their intersection value. Sorting ensures fast detection of nearby intersection and fast search for specific intervals.

The first aim of this step is merging all nearly coincident and redundant intersections belonging to different input meshes. The second aim is arrange the data structures in Marching Cubes [16] compliant manner. Arranging the data structures in a MC-compliant way means to ensure the existence of no more than one intersection for each virtual cell edge. To achieve this, the data structures *XY, XZ* and *ZY* are processed iteratively by two main functions: fuse and removal actions.

Even though the intersection data structures do not explicitly represent a grid, it is quite easy to reconstruct the virtual cells of the grid by simply locating the corresponding edges in the *XY, XZ* and *ZY* data structures.

The two basic operations that allow us to pursue the goal of arranging the data structures in a MC-compliant way are the *fusion* and r*emoval* actions.

4.3.3.1. Fusion operation

The fusion operation performs the proper merge step of redundant intersections. The fuse operation takes part between consecutive pairs of intersections in the same grid line. The algorithm takes the decision to fuse a couple of intersections according to the maximum merge distance defined by the user. This value should be fixed carefully in relation with the registration error or the resolution of our acquisition device.

Two intersections can be fused if they belong to different meshes, have concordant signs and their distance is shorter than the maximum distance. The distance between two intersections is defined by the formula:

$$d(i1,i2) = \frac{|ic_i1 - ic_i2|}{dr(i1,i2)} \leq maxdist \tag{4.3}$$

where *dr(* i_1, i_2*)* is:

$$dr(i,j) = \frac{w_i dr_i + w_j dr_j}{w_i + w_j} \tag{4.4}$$

In the fuse process, we can find two kind of situation. The first one is the one where both intersections belong to the same virtual cell edge. The second one is when the couple of intersections belong to different virtual cell edges.

If two intersections can be fused and belong to the same virtual edge, both of them are erased of the data structured. A new one representing the fused intersection is added. The intercept (*ic*) value of the new intersections is the weighted average of the previous ones. The mesh name (*nm*) is the combination of the previously ones and the sign remains constant as before. The quality field is updated by adding the both weights.

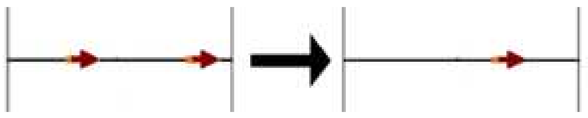


Figure 11: Merging between two intersections belonging to the same edge.

If two intersections belong to different virtual cells, the fusion operation performs a shift of the intersections towards the new average location. This implies that one of the intersections crosses from one virtual cell to next. Whenever we cross from one virtual cell to another one, a couple of artificial intersections are created and inserted in all the perpendicular edges the intersection touches.

The artificial intersections are initialized with the intersect (*ic*) value undefined, as we know that there is one intersection in this edge but we do not know exactly its position. The mesh name (*nm*) is equal to the intersection that generated it (the one that cross to the new cell). Its sign is according to the generator intersection. Finally direction and weight are set to null.

These artificial intersections are added in order to ensure consistency of data structures by simulating a discretization step (detection of the intersections with the grid line). The Figure 12 shows an example of a fusion of two intersections belonging to different virtual edges

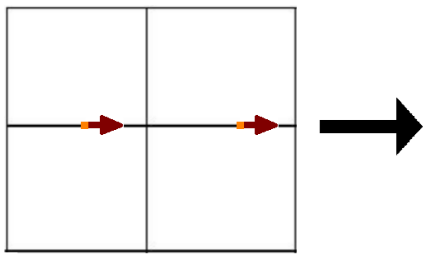


Figure 12: Merging between two intersections belonging to different edges.

4.3.3.2. Removal operation

The removal operation is performed in all grid edges before fusion process start and on the currently updated grid edge, for example, due to a new added artificial intersection.

During the initial removing phase, the entire intersection lists are analyzed. A removal action is performed on each pair of intersection that lies in the same virtual cell. A couple of intersections are removable if they belong to the same mesh and have discordant signs. This corresponds to the removal of high quality details in a range map.

When a couple of intersections are removable, we delete the bit that refers to the mesh that they have in common. The intersections are removed from the data structures if their field mesh name (*nm*) equals zero. As this, means that they do not belong to any mesh anymore.

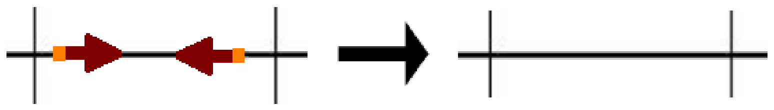
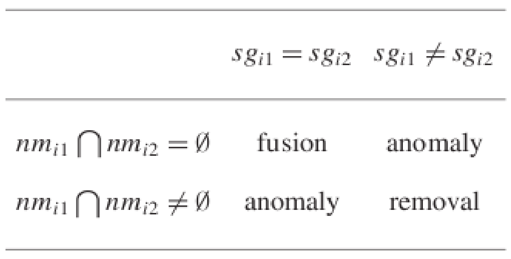


Figure 13: Removal operation.

Most of the artificial intersections are erased in the following removal check process. From the remaining artificial intersections, we remember that they are initialized with their intercept field value set to unknown. At this time, we need to assign an intercept value. We calculate his intercept value via linear interpolation with their neighbor intersections.

Unfortunately, the fusion and removal operations are not sufficient to ensure that no more than one intersection lies on each virtual edge. In the cases in which anomaly is detected come from possibly small malformations of the original meshes. See Figure 14:



	$sg_{i1} = sg_{i2}$	$sg_{i1} \neq sg_{i2}$
$nm_{i1} \bigcap nm_{i2} = \emptyset$	fusion	anomaly
$nm_{i1} \bigcap nm_{i2} \neq \emptyset$	anomaly	removal

Figure 14: Merge situations.

If one edge has more than one intersection, our algorithm solves the ambiguity keeping the one with higher weight value and deletes the other ones. This is done because it is required by the surface reconstruction step. The presence of more than one intersection in an edge produces cell failures.

4.3.4. Surface Reconstruction

In this section, we briefly discuss the concepts related Marching Cubes reconstruction Next, we detail the Marching Intersection surface reconstruction approach.

4.3.4.1. Marching Cubes approach

Marching Cubes [16] is a volumetric algorithm for creating a polygonal surface representation of an isosurface of a 3D scalar field. It combines simplicity with high speed since it works almost entirely on look up tables.

First, volumetric space is subdivided into a series of small cubes (voxels). Each cube is defined by its eight vertices, where each vertex is assigned a scalar value of *zero* or one whether the vertex is located outside or inside the model. As a result, there are 256 (2^8) possible configurations. Each configuration represents a way that the surface can intersect the cube.

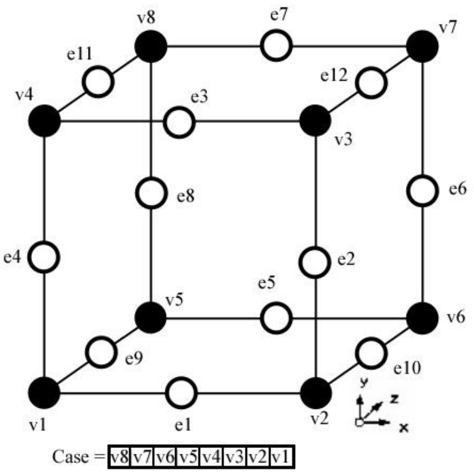


Figure 15: Marching Cubes vertex and edge convention.

The Marching Cubes algorithm provides a look-up table of 256 entries that define all the possible ways that a surface can intersect the cube. For each of the possible vertex states listed in this table, there is a specific triangulation set of the edge intersection points. Each entry of the table defines the triangulation patch by a maximum number of 5 edge triples with the entry terminated by the invalid value -1. Here you have two rows of the look-up table; the both entries codify the triangulation patch only with one triangle:

```
int triTable[256][16]{
        {1, 4, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}
        {1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}
        ...
}
```

Of the 256 possible configurations, the symmetry reduces it to 15 different patterns. The Figure 16 shows the 15 different resulting patterns:

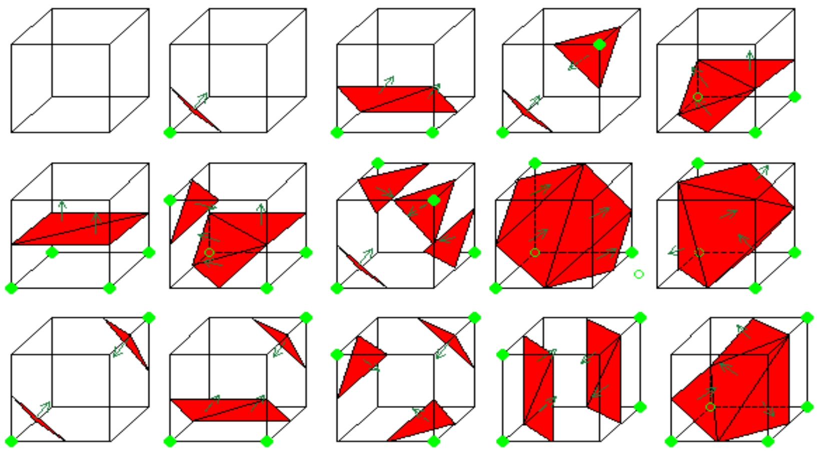


Figure 16: Marching Cubes reconstruction patterns.

As a summary, for every cube in the grid, we examine the values at their eight corners and determine active edges (the edges with an intersection) of the cube. Next, a look-up table indexed by a bit binary code configuration of the eight cube corners that yield the topology of the surface inside that cube. After processing each cube in the grid, the surface is complete.

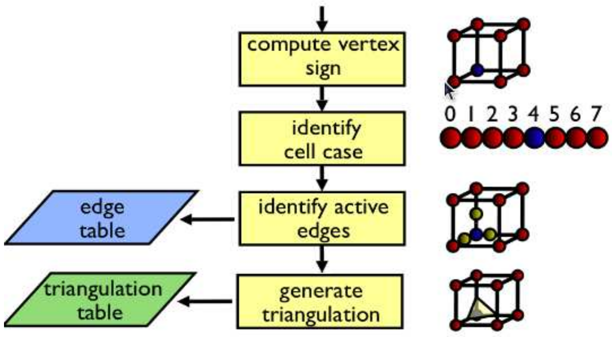


Figure 17: Marching Cubes overview.

4.3.4.2. Marching Intersections approach

At the end of merge process our data structures contain the signed intersections between the integrated surface and the lines of reference grid. So our algorithm, Marching Intersections, represents a slightly difference in the way that we classify the vertices in respect with Marching Cubes. The way that the surface cut the cube is completely defined by the signed intersections of the grid lines (edges). However, we can use the signed intersections information to infer the status of the corners.

Even though the intersection data lists do not explicitly represent a cube, it is quite easy to reconstruct the virtual cells of the grid by simply locating the corresponding edges in the *XY, XZ* and *ZY* data structures.

The classification of the vertices of a virtual cell can be obtained easily from the analysis of the existing intersections in the edges. If and edge contains an intersection, those vertices classification, interior / exterior, just depend of the sign of the intersection. If one edge has an intersection, the classification of those vertices is opposite, one inside and the other outside. In the other hand, if one edge has no intersection, his vertices classification are concordant, the both interior or both exterior.

We are going to show how reconstruct the surface inside one cube. Assume the cube of the Figure 18 with four edge intersections. With the sign and direction of the four intersections we are able to determine which vertices are inside / outside the surface. After the classification of the vertices, we know the 8 bit binary code that indicates which of the vertices is inside / outside and checking in the MC look up table, we obtain the corresponding triangular patch for this vertex configuration.

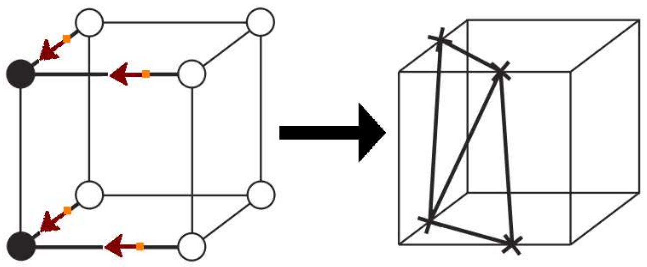


Figure 18: Vertices classification and surface reconstruction.

It can also happen that, we are not able to determine the classification of the vertices. If that happens, it means that we have an incorrect cell configuration. We are in that situation when the set of cube edges intersections create contradictions in the vertex classification procedure. For example, in the case one intersection indicates that one vertex is outside the surface, while at the same time, another intersection indicates is inside. Such an inconsistency is shown in Figure 19:

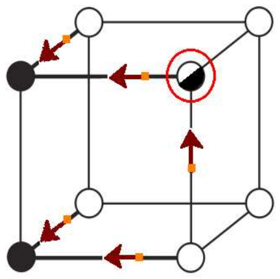


Figure 19: Wrong cell configuration.

This configuration is wrong because according to the intersections that we have in the cube, we are not able to classify the circled vertex due to there is intersection ambiguity. So, as we cannot define a configuration, we cannot search the look-up table for a triangulation.

As a result, we have a hole in the reconstructed model. In the following chapters we will discuss how we handle these situations.

We implemented an iterative algorithm visiting the entire intersections list. This algorithm is designed efficiently. We only visit each virtual cell once.

We visit in sequence the three data structures, *XY, XZ* and *ZY*. For each virtual cell that has an intersection in the *XY* data structure we compute the triangulation. For example, if our data structure *XY* in the position (2, 3), it means, *x* value equal to two and *y* value equal to 3, has the following intersection:

```
struct Intersection {
            float ic = 5.3;
            int nm = 3;
            bool sign = 1;
            float dr = 0.89;
            float weight = 1;
            int meshPosition = -1;
    };
```

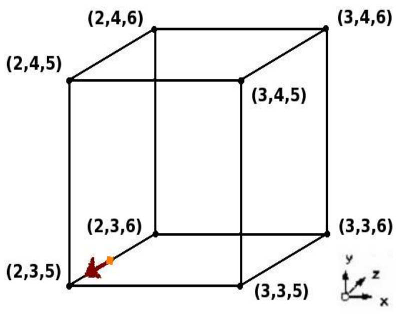


Figure 20: Virtual cell reconstruction from the reference grid.

The computed cell is the one corresponding to (2, 3, 5). Furthermore, the other cells that share the edge that has the intersection are also check. If they were not compute previously, we compute them at this point. In this case, cells (1, 3, 5), (1, 2, 5) and (1, 3, 5) will be also computed.

After finishing the *XY* loop, we apply the same procedure to each virtual cell that has an intersection in XZ and was not compute in *XY* loop. And again for all cell that have intersections in *ZY*. As we can see, we do not have to maintain a trace of the processed cells because we adopt a visiting strategy.

For each successful cell, we have all the needed information to reconstruct the surface inside. We have the intersections, and we have the corresponding triangulation according to the look up table of Marching Cubes. So our intersections now become the vertices of the triangles, and the triangles become faces in the integrated mesh.

At this point, we are able to do a test to analyze what comes out of our application. We tested it on three different objects. In Figure 21, the model of the left correspond to the fusion of twelve meshes, the model of the center and the right where created by fusing nine meshes.

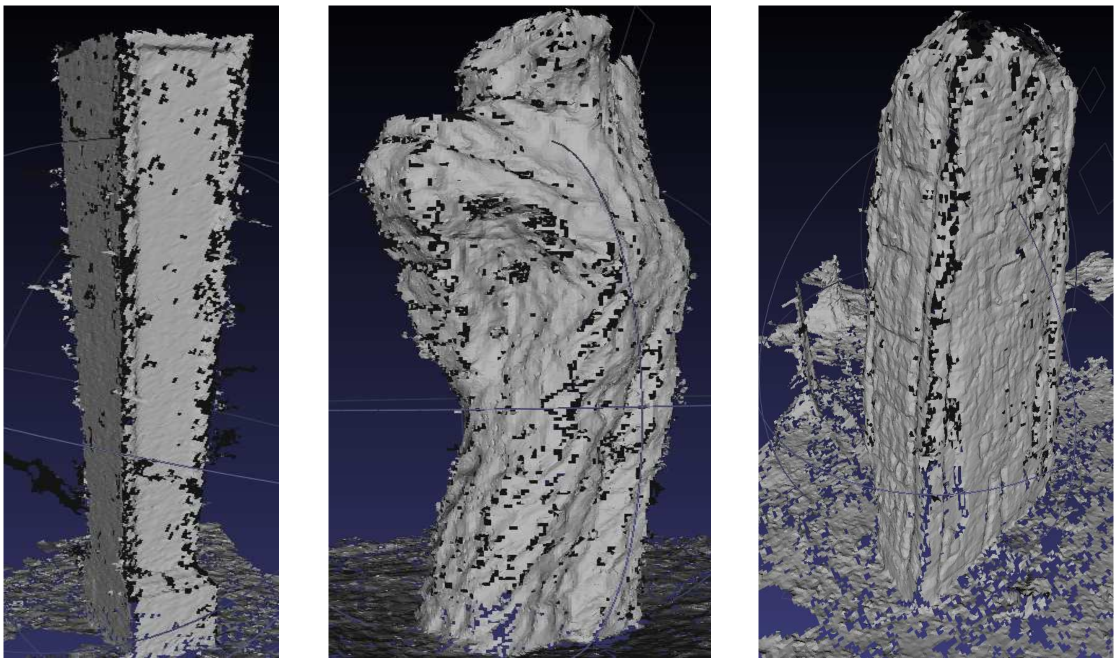


Figure 21: Marching Intersections surface reconstructions samples.

As we can see, the final models are plenty of holes. This means that a lot of cells have wrong configurations leading to failed reconstructions. The result is not good enough, so we need to figure out some techniques to improve the final 3D model. The next chapters are related to the applied techniques for better results.

4.4. Improving techniques

Three separate techniques have been developed where each one affects a different stage of the pipeline:

The first one is applied before the integration, and is tries to improve the quality of the input meshes in order to reduce the number of wrong cells configurations due to the merging process. We observe that the meshes with noise, the mesh boundaries and sharp features of the object, are a source of errors in the surface reconstruction. For that, some clean up and filtering techniques are applied to the input meshes.

The second one is applied during execution time, and helps Marching Intersections algorithm to get better results. We observe that we have a lot of wrong cell configurations in surface reconstruction step. This is because, after the merging process between meshes, we obtain cell configurations that fail. This occurs because there are intersections that should not be there. One proposed method is to assign different confidence or quality to each intersection. In case of a cell failure, we can decide which intersections are more confident and which ones do not, and try to compute again the cell.

The third one is computed after the surface reconstruction process finish and tries to fix what the other two previously methods cannot. We examine the integrated mesh by detecting holes in the integrated surface and trying to close them. For this, a mesh based hole-filling algorithm is chosen. One of our requirements is that our integration algorithm should be fast. In the literature there is a lot of filling holes algorithm. It is a difficult subject due to the huge amount of situations and type of holes. Given the trade-off between speed and robustness; we prioritize speed of the algorithm before robustness.

In the following sections we give more details about these three techniques and they way they were implemented.

4.4.1. Filtering of the input meshes

The aim of this step is to clean up the mesh in order to delete all the noisy and inaccurate data that are source of error in the integration step. In such a way that, the resulting filtered mesh has the most reliable data.

It consists of two main phases. In a first step, we clean-up the mesh with the aim of deleting noise and high frequency details. Next, we close some small holes inside the depth maps with a diffusion procedure. Finally we erode the boundaries as edges are typical less reliable. In a second phase, apply some filtering techniques in order to smooth the surface.

4.4.1.1. Mesh clean-up process

For cleaning up the mesh, we create a mask. The goal is to create a mask that includes the parts of the object that we want to keep and to superimpose it on the depth map. In this way, we can generate a new partial mesh with only the areas marked in the mask, removing all undesired parts. The best desired mask would be uniform, without holes and without small floating blobs mask. We detail the steps to create this mask:

To compute a mask, we need to use the respectively depth and count maps of the image. We decide a minimal count value according our requirements; the count value decision is a trade-off. A higher count value results in a cleaner but smaller mesh (there is a possibility that if the count value is too high we lose useful data). A lower value will result in a more noisy mesh, but without losing data. After fixing a minimal count value, we check for each pixel on the depth map its corresponding count value. Only the pixels with at least the minimal count value will be included in the mask.

In the Figure 22 on the left we have a depth map file, and on the right the resulting mask after applying a minimal count value equal to *four*. We can observe that the mask still has some non-uniform areas and some floating parts that may difficult the reconstruction.

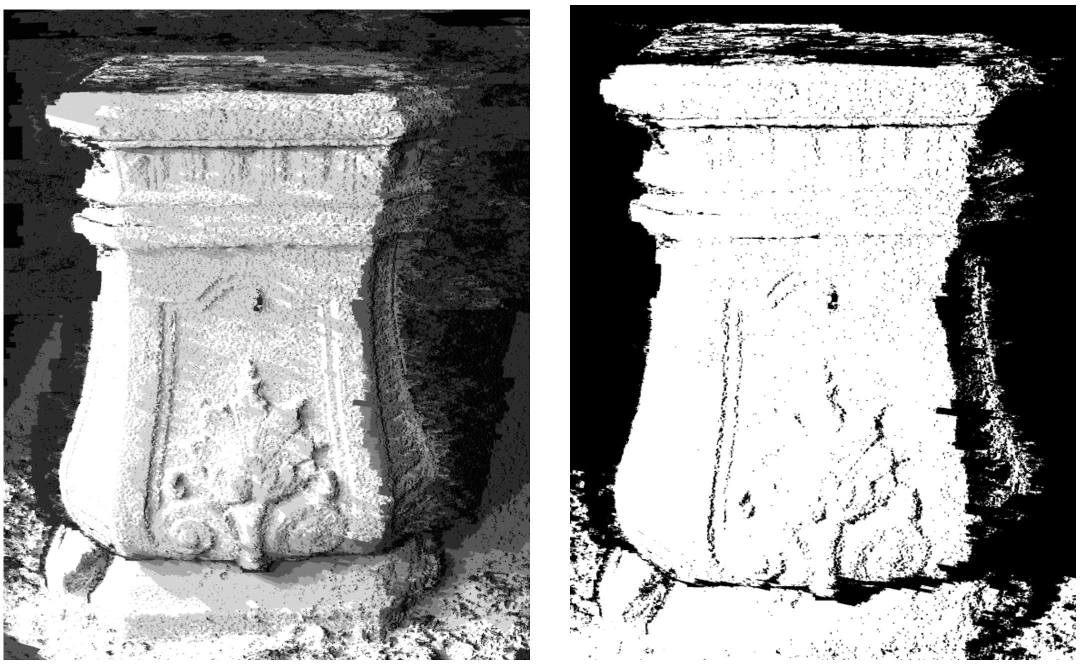


Figure 22: Left: Depth map. **Right:** Resulting mask after applying minimal count criteria.

Secondly, we would like to remove these floating parts of the mask, because it makes the mesh noisy and produce quite a lot of problems (cell configuration failures) in the fusion step with other meshes. For this reason, blob detection is computed; with this step we want to detect the main blobs and delete the floating parts.

In the Figure 23, the current mask is shown on the left and on the right, the mask after the blob detection applied. The red area is the main blob detected. The red part will be kept while the white parts will be deleted from the mask after this step.

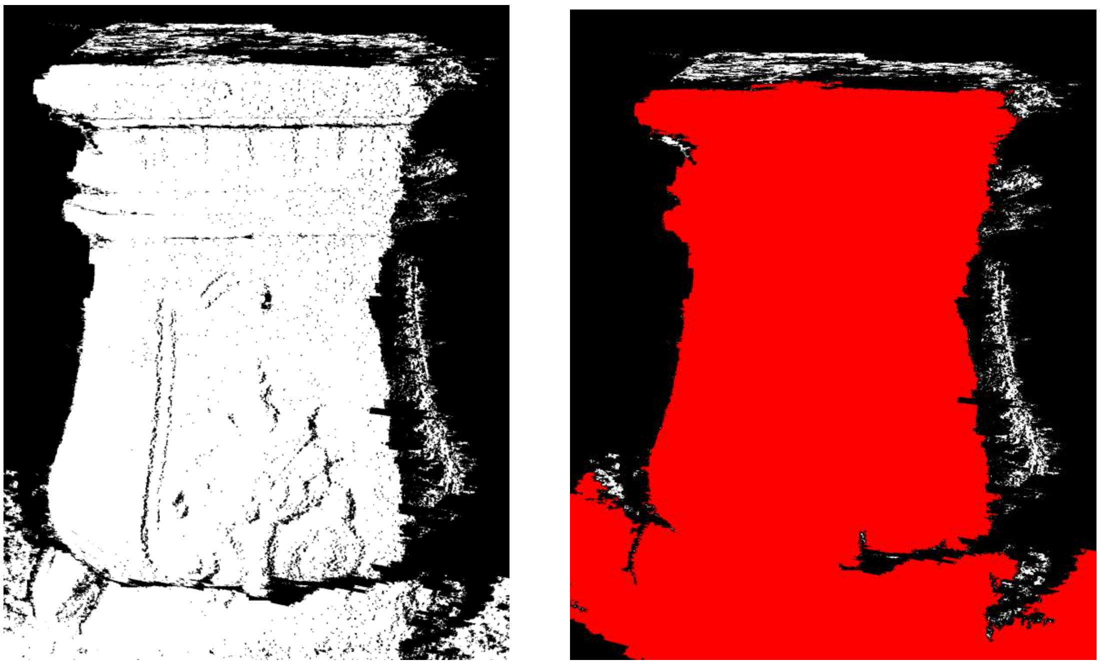


Figure 23: Left: Mask. **Right:** Blob detection procedure.

Now we can see that the mask is uniform, but we have a plenty of small black dots in the image. To close them, we apply dilate / erosion image procedure. With the dilation procedure we add pixels in the boundary areas of the mask until we close the small holes in the mask (black dots). With the erosion procedure, we remove pixels from the mask boundaries. In this process, we apply higher erosion than dilate, because we want to remove the original boundary pixels from the mask. Boundary pixels are generally less reliable and normally produce cell failures.

We take in consideration that, the black dots of the original mask do not have valid depth data. Then, if we want to include these pixels in the mask, we need to compute before a valid depth value for them. This is done by diffusing the valid depth values from neighboring pixels onto these pixels that do not have already a valid depth value.

At this point, we achieved our goal; we have a uniform mask without floating parts and without holes. See Figure 24:

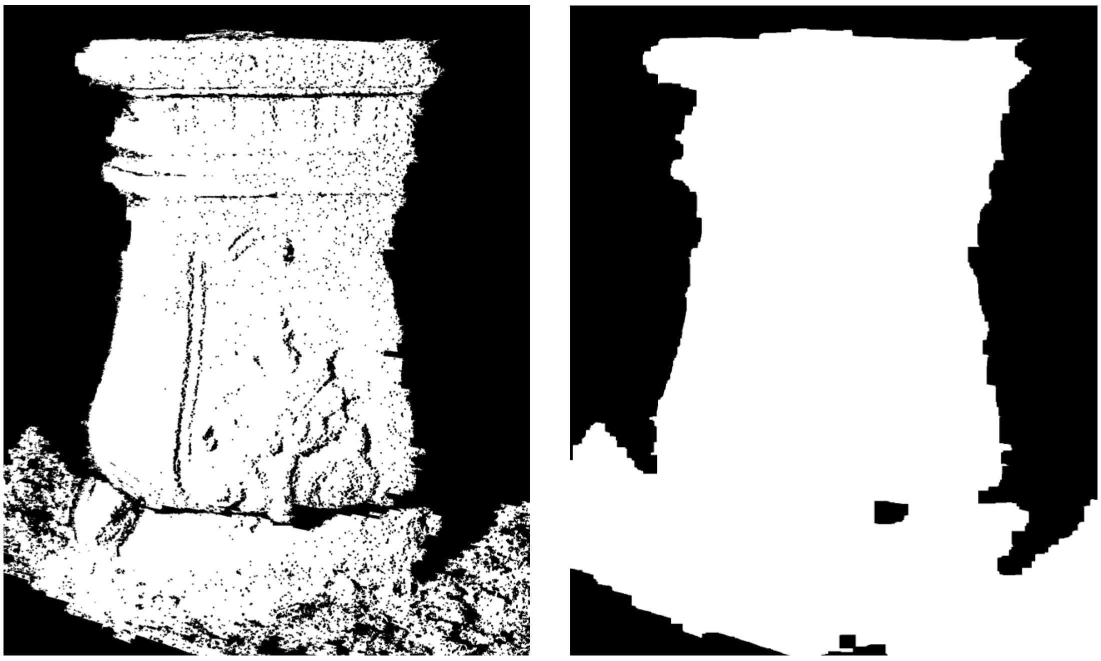


Figure 24: Left: Mask after blob filtering. **Right:** Resulting mask after erosion / dilation process.

During this clean-up filtering we use some screenshots about how we create the mask step by step. The concrete parameters that we used to clean-up this model are the following:

- Count map Threshold = 4
- Number of pixels of dilation = 14
- Number of pixels of erosion = 18

Note that, the dilation / erosion values are high. In Figure 25 on the left we can observe that, there are quite black areas that belong to the surface. We needed a high dilation value to fill all the black areas that we want to keep. So consequently, the erosion value is also high. The erosion value is higher than the dilation value to erode the original depth boundaries.

4.4.1.2. Bilateral Filtering

The second phase of the filtering step is a process of surface smoothing. In the broadest sense of the term "filtering", the value of the filtered image at a given location is a function of the values of the input image in a small neighborhood of the same location. For example, the Gaussian smoothing filter is a 2D convolution operator that is used to `blur' images and remove detail and noise. The degree of smoothing is determined by the standard deviation of the Gaussian function.

It computes a weighted average of pixel values in the neighborhood, in which the weight decrease with distance from the neighborhood center. Although, formal and quantitative explanations of this weight fall-off can be given, the intuition is that images typically vary slowly over space, so near pixels are likely to have similar values. Therefore, is appropriate to average them together.

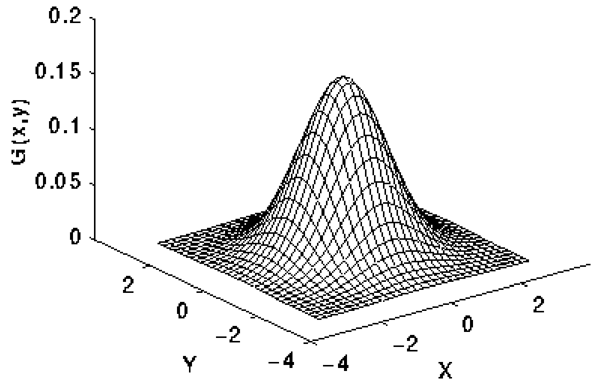


Figure 25: Gaussian distribution with mean (0, 0) and σ=1.

The noise values that corrupt these nearby pixels are mutually less correlated than the signal values, so noise is averaged away while shape is preserved. The assumption of slow spatial variations fails at edges, which are consequently blurred by linear low-pass filtering.

How can we prevent averaging across edges, while still averaging within smooth regions? Many efforts have been devoted to reducing this undesired effect. Bilateral filtering [17] is a simple, non-iterative scheme for *edge-preserving smoothing*, by means of a non linear combination of nearby image values.

Two pixels can be close to one another, that is, occupy nearby spatial location, or they can be similar to one to another, that is, have nearby values, possibly in a perceptually meaningful fashion. Closeness refers to the vicinity in the domain, similarity to vicinity in the range. Gaussian filtering is a domain filtering, and enforces closeness by weighing pixel values with coefficients that fall off with distance. The idea underlying bilateral filtering is to do in the range of an image what Gaussian filters does in its domain.

Spatial locality is still an essential notion. In fact, we can combine then range and domain filtering and show that the combination is much more interesting. Actually, we denote the combined filtering as bilateral filtering [17].

Similarly, we define range filtering, which averages image values with weights that decay with dissimilarity. Range filters are non linear because their weights depend on image intensity or color (this defines a similarity function between nearby pixels). Most importantly, they preserve edges that are one feature that really interested us in our system.

Bilateral filtering [17] replaces the pixel value *x* with an average of similar nearby pixel values. In smooth regions, pixel values in a small neighborhood are similar to each other. The similarity function is then close to one. As a consequence, the bilateral filter acts here essentially as standard Gaussian filter, and average away the small, weakly correlated differences between pixel values caused by noise.

Consider now a sharp boundary between a dark and bright region. When the bilateral filter is centered, say, on a pixel on the bright side of the boundary, the similarity function assumes values close to one for pixels on the same side, and close to zero in pixels in the dark side. The similarity function is show in Figure 26 in the center.

As a result, the filter replaces the bright pixel at the center by an average of the bright pixels in its vicinity, and essentially ignores dark pixels. Conversely, when the filter is centered in the dark pixel, the bright pixels are ignored instead. Thus, as shown in Figure 26, good filtering is achieved at the boundaries thanks to the domain component of the filter, and crisp edges are preserved at the same time, thanks to range component.

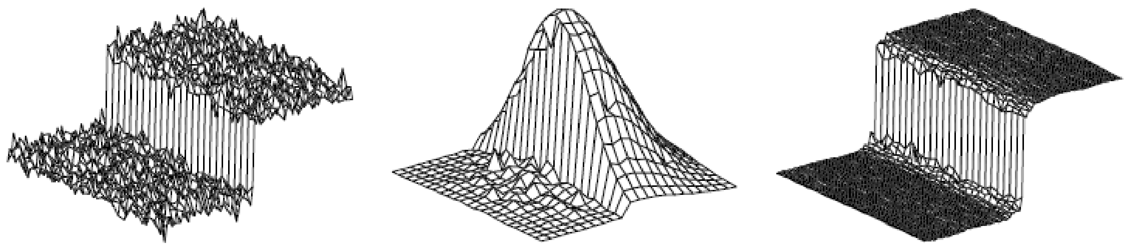


Figure 26: Left: A sample noise surface. **Center:** A bilateral distribution function. **Right:** Resulting filtered surface.

Now that we understand how a bilateral filtering works, we can try and show what comes out after applied it to one depth map. Our input images are the depth maps that at the end can be seen like a grey scale images where the intensity defines the distance from the camera, in other words, the 3D dimension. So according to it, we can apply the bilateral filtering with guaranties to success. Filtering the same model used previously, we can see in Figure 27 on the left the original mesh and on the right the mesh after being filtered by bilateral filtering. We can observe that the surface is less noisy and more smoothed. At the same time, the edges and boundaries that give detail and shape to the object are kept.

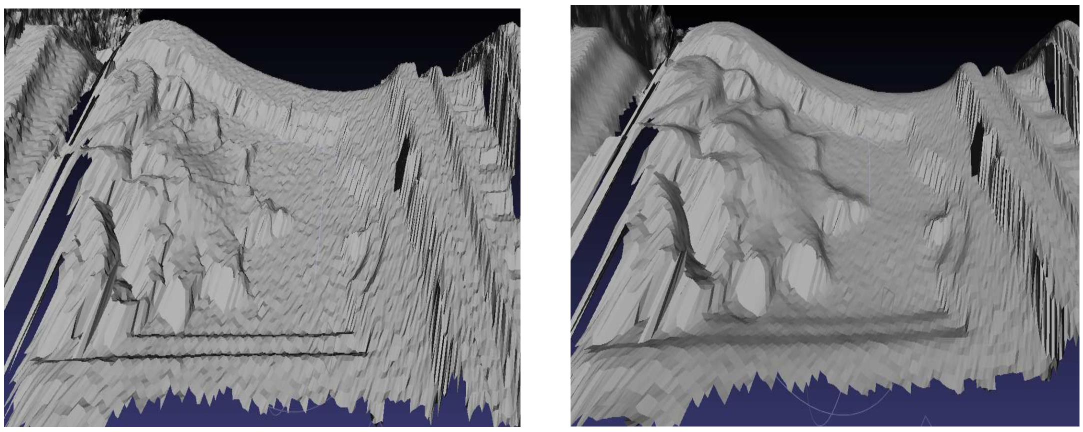


Figure 27: Left: Sample mesh. **Right:** The resulting mesh after a bilateral filtering.

As before computing the mask, there is another trade-off here. The stronger the filtering, the smoother surface will become, resulting in an easier reconstruction. But in the other hand, this reconstruction will be less detailed due to the loss of information in the filtering. Is therefore important, to study each case and choose the most appropriate election according of the shape of the object and the required quality reconstruction.

At this point, the filtering step is finished and we can compute the resulting filtered mesh by combining the filtered depth map, the camera parameters and the mask that we computed before. For each depth filtered map pixel we check if is in the mask. If it is, we compute the 3D projection with the camera parameters and the depth map. If the pixel is not in the mask, this point is not added to the new mesh.

The Figure 28 on the left is the original mesh without cleaned up and neither filtering, and on the right, the mesh after applying the clean up and filtering. We observe that the floating and less confident parts are deleted. Also, we can appreciate the effects of the bilateral filtering, the noise is removed and the sharp areas smoothed.

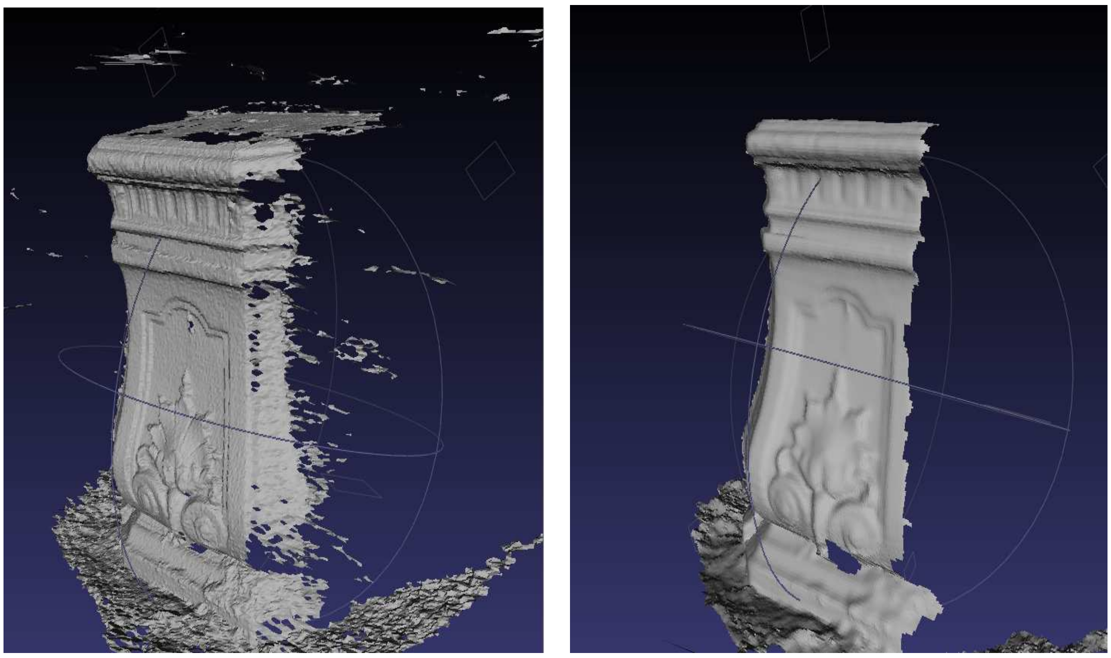


Figure 28: Left: Sample mesh. **Right:** The resulting mesh after a process of clean up and bilateral filtering.

During this bilateral filtering process we these concrete parameters:

$\sigma_s = 10$
$\sigma_r = 0.005$

Where σ_s is the standard deviation in the domain filtering (Gaussian filter) and σ_r is the standard deviation in range filtering.

4.4.2. Weighted Intersections

We described before the intersection data structure of our system, where in each intersection a field called *weight* is defined that until now was set by default to one. We will use this field to assign quality to each intersection. In this way, a more confident intersection will have a higher *weight* value than another one less confident. We determine the confidence of one intersection using the camera position and the quality count value of the intersection.

If we use the camera position, we can consider that one point is more confident if it is seen more orthogonally with respect the camera view, and less confident if it is less orthogonal. We can check the orthogonality checking the angle between the vector between to the camera position and the target vertex and the vertex normal.

Furthermore, one of the consequences of non-orthogonality is that in the mesh we obtain skinny and long triangles. For example in the Figure 29, we have an image and his corresponding mesh representation where we can see this fact. The long skinny triangles match with the points that are not well-seen for the camera.

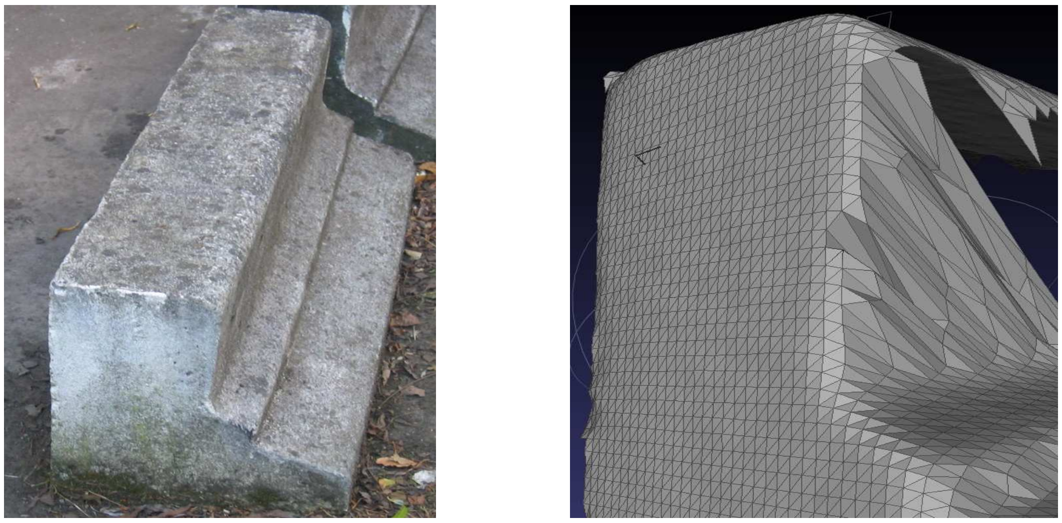


Figure 29: Left: Original image. **Right:** Corresponding mesh representation with skinny triangles where the camera angle is not orthogonal.

Another approach that we can take is in the discretization step. We can check the view angle between each normal face and the camera position and define a minimum threshold. If a face view angle is less orthogonally that the defined threshold, this face is not included in the grid.

Related to the count value: if one point has a higher count value (means that is seen for a higher number of more cameras), we can consider that is more confident that another one seen for less count value.

A last approach is taken in consideration. The weight field of intersections generated by the boundaries of the meshes is set to the lowest value. We observe that, boundaries are a source of cell failures.

Finally, the resulting weight assigned to the intersection is the product between the camera view angle and the count value, unless it is a boundary intersection that, is set to the minimum.

4.4.3. Hole-filling algorithm

After evaluation of the current pipeline, we noticed that, the reconstructed models still contained many holes. This showed the necessity to implement a hole filling algorithm. The hole-filling algorithm is computed after the MI reconstruction finish. The first step is to detect the holes in the mesh surface. The second step is, try to fill the maximal number of them in a proper way, that is to say, without self-intersecting faces.

Before we start explaining how we detect the holes in the integrated surface, we give some definitions about some related concepts [18].

4.4.3.1. Basic concepts.

A triangular mesh is defined as a set of vertices and a set of triangles that joint these vertices. If two triangles share a common edge, the two triangles are adjacent. An edge usually links two triangles. If not, the edge is called a *boundary edge*. A hole is a closed loop of *boundary edges*. *Boundary triangles* are those triangles that own one or two *boundary vertices*.

All triangles who share one common vertex are called the *one-ring triangles* of the vertex. All edges who share one common vertex are called the *one-ring edges* of the vertex. And, all the vertices on *one-ring edges* of a vertex (except itself) are called the *one-ring vertices* of the vertex. A *non-manifold* edge is an edge that has more than two adjacent triangles. An interior edge is an edge that is not a boundary edge. A singular vertex is a vertex that has more than two adjacent *boundary edges*. A *manifold* mesh has no *non-manifold* edges and no singular vertices. In Figure 30 we show graphically the previous concepts.

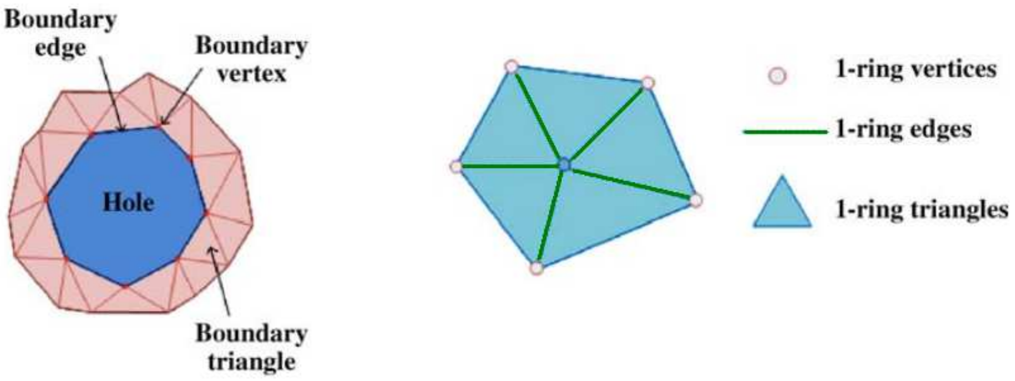


Figure 30: Related hole detection concepts.

All mesh models are assumed to be oriented, manifold and connected, and a given hole is assumed to have no islands (an island means to have triangles inside the hole). To efficiently support our hole filling algorithm, a vertex-based topological structure is used, which records *one-ring vertices*, *one-ring triangles* and *one-ring edges* of every vertex of the reconstructed model.

4.4.3.2. Hole identification

The first step of hole-filling is to detect all the holes in integrated triangular mesh of the model. The principle that we use to identify the holes is that boundary vertices can be easily identified by checking the numbers of their *one-ring triangles* and *one-ring edges*. If the number of *one-ring triangles* and *one-ring edges* of a vertex are not equal, it means that this vertex is a boundary vertex.

To achieve this, it is necessary to have full knowledge of each vertex inside the integrated mesh. For each vertex we need at least to know which are their *one-ring vertices,* their *one-ring edges* and which of them are boundary edges and which are not.

Therefore, in the surface reconstruction step, a dynamic data structure is created for every new vertex added to the integrated mesh. This data structure has the following fields:

```
struct VertexData {
        int ring_edges;
        int ring_triangles;
        vector <int> adjacentVertexPos;
        vector <bool> boundaryEdge;
        int label;
        }
```

Where:

ring_edges: Is a counter of the number of one-ring edges has the vertex.

ring_triangles: Is counter of the number of one-ring triangles has the vertex.

adjacentVertexPos: Is an array of the locations of the adjacent vertices has this vertex.

boundaryEdge: Is an array of Booleans indicating for each one-ring edge of this vertex if is a boundary or non boundary edge.

label: Is one auxiliary field used for detecting the main blobs of the integrated surface. Each interconnected vertices by at least one edge will have the same label value, consequently, each vertex located in the same blob will have the same label value. In this way, the floating parts can be removed from the final model.

During the execution of the surface reconstruction step, an array of VertexData data structures is filled. Remember that, the Marching Intersections algorithm returns for each correct cell, a patch of the surface model defined by a set of triangles. These triangles have as vertices the intersections with the grid reference lines. Consequently, for each new triangle, we must update properly the corresponding vertices data structure.

We update the data structure in the following way. We need to increment the field *ring_triangles,* for each vertex of the vertices has the triangle. The field *ring_edges* must be incremented only if this edge was not counted previously (it means is a new edge of the vertex) and the field *adjacentVertexPo* correspond to the array position of the edge vertex in the integrated mesh. Finally, we can determine if an edge is a non-boundary if its edge is shared by two triangles.

At the end of the surface reconstruction process, we have filled an array of VertexData data structures which size equal to the number of vertices has the integrated mesh. The hole detection process can now start.

A sequential loop visits all elements of the array VertexData structure. For each vertex, we check if this is a boundary vertex. Once a seed boundary vertex is identified, from it, a set of connected boundary edges can be traced from it. This is the point where it is useful know which edges of the vertex are boundary edges. We only try to trace a hole using the neighbor edges that we know are boundary edges. Furthermore, a mark is set at each boundary edge used in hole detection in order not to use it again or in case of holes that share one common vertex. If all the boundary edges form a closed loop, they make up a hole.

In this way a hole in the integrated mesh is identified. All identified holes are stored in an array of holes data structure. A hole, at the same time, is defined by an array of the vertices that compose the hole:

```
struct Hole {
        vector <int> boundaryVertexPos;
        }
```

4.4.3.3. Hole-filling overview

Ideally, hole-filling algorithm should posses the following properties:

-Able to cover an arbitrary hole (robustness).

-Capable of filling large holes in a reasonable amount of time (efficiency).

-Enable the patched surface to match the missing geometry well (precision).

Unfortunately, due to the complexity and diversity of the holes, no existing hole-filling methods satisfy all the above desirable properties. In particular, the robustness is hard to achieve. Many hole-filling approaches have been proposed in the literature. These approaches can be dividing into two categories: voxel-based and mesh-based.

In the voxel based approach, a mesh model is first converted into a volumetric representation which consists of discrete volumes named voxels. Then different methods are utilized to patch the holes in volumetric space. Davis [19] used volumetric diffusion to fill the gaps. Curless [3] employs space carving and isosurface extraction to fill holes. Joshua and Szymon [27] use a min-cut algorithm to split space into inside and outside portions, and patch the holes simultaneously in a globally sensitive manner.

Voxel-based approaches work well for complex holes but they are all time-consuming and may generate incorrect topology in some cases.

In mesh-based approaches, the holes are patched by dealing with the triangles directly. Holes with regular boundary over a relatively planar region can be easily patched via planar triangulation. However, filling a complex hole over an irregular region is much more difficult. To solve this problem, Carr [20] used radial basis function (RBF) to construct an implicit surface to cover the hole. This method works well for convex surfaces and can handle irregular holes. But difficulties arise when the underlying surface is too complex to be described by a single-value function. Liepa [21] described a method for filling holes in unstructured triangular meshes. The resulting patching meshes interpolate the shape and density of the surrounding mesh.. Jun [22] proposed a hole filling method based on piecewise scheme. His method divides a complex hole into several simple holes and all sub-holes are sequentially filled with planar triangulation. Sub-division and refinement are then employed to smooth the new triangles. The negative side of the method is that, too many overlaps or twist may make it crash and iterative refinement is a time consuming process. Chen [28] proposed a hole-filling method which can fill the hole and recover its sharp feature involved in the hole area. With this method, holes are filled using a radial basis function, a feature enhancement process based on Bayesian classification and sharpness dependent filter is then applied if there exists any sharp feature on the hole boundary.

As a summary, most existing methods have difficulties in dealing with complex and highly curved holes.

4.4.3.4. Hole-filling implementation

The speed is one of the requirements of our application. For that reason, we discard voxel-based hole-filling approaches and we choose a mesh based approach. Although, mesh based approaches have difficulties dealing with complex holes. We assume that we will reduce and simplify the number of holes in the integrated mesh and the approach will be good enough.

Specifically, we chose a recursive loop splitting hole procedure [23]. First of all, polygonal holes can fall into two categories, simple and complex holes. The holes denoted as simple are the ones that whose boundary edges can be projected on a plane without self-intersections. In the other hand, due to shape complexity, some holes projected on a plane have self intersection boundaries edges. In Figure 31 we show a graphical example of simple and complex holes.

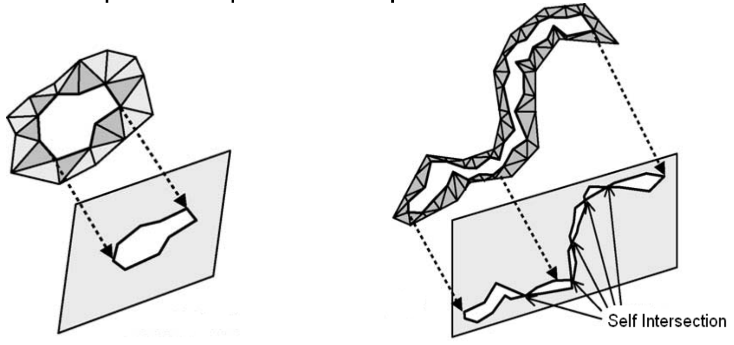


Figure 31: Left: Simple hole. **Right:** Complex hole.

As far as possible, we must avoid complex holes as the algorithm will not be able to fill it or will instead, produce a patch that produce self-intersecting faces. Previously to compute the hole-filling algorithm, we need to project every hole in a plane. The projected plane is defined by its normal vector. In Figure 32 on the left, a hole projection in a plane P.

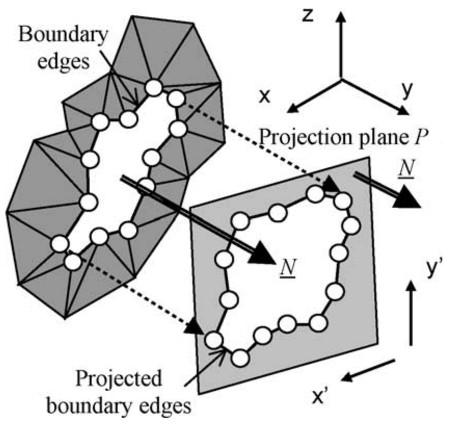


Figure 32: A hole planar projection in plane P.

At this point, we have projected the hole boundary edges in a plane, called now on as average plane. Our hole-filling algorithm is a recursive loop splitting hole procedure. Each projected hole to be triangulated is divided in two halves recursively, until each half has three vertices. A hole of three vertices, actually, represents a triangle that we can reconstruct and add to the mesh.

The division is along a line (split line) defined from two non-neighboring vertices in the hole. We define a split plane as the plane orthogonal to the average plane that contains the split line. You can notice that, there are multiple splitting choices, but not all the possibilities can be use. One split line division is acceptable only if the two halves form two non-overlapping loops.

In order to determine whether the split forms two non-overlapping loops, the split plane is used for a half-space comparison. That is, if every vertex in a candidate half loop is on one side of the split plane, then the two loops do not overlap and the split plane is acceptable. We determine in which side of the split plane are the vertices computing the distance to the plane and checking the sign of this value.

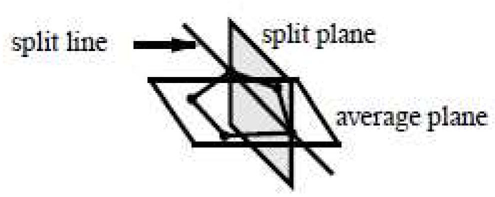


Figure 33: Hole division via split plane.

Typically, however, each loop may be split in more than one way, to achieve higher percentage of successful reconstructions is quite important to select the optimal splitting plane. Although, many possible measures are available, we have been successful using a criterion based on aspect ratio. The aspect ratio is defined as the minimum distance of the two halves loop vertices to the split plane, divided by the length of the split line. The best split plane is the one that yields the maximum aspect ratio. This splitting procedure is not perfect and can fail in some hole triangulation.

We keep computing the split plane of every sub hole recursively until every spitted hole has a size of three vertices. In this way, the entire hole is triangulated.

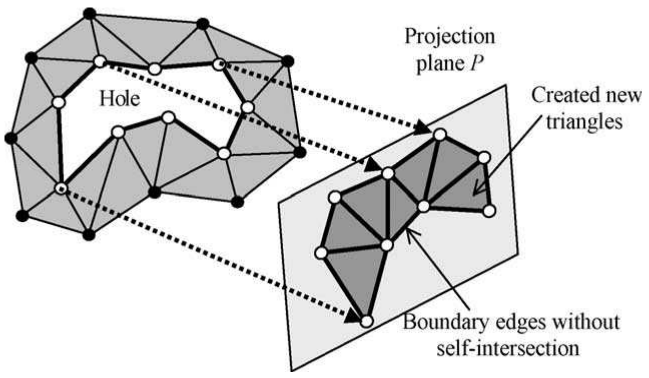


Figure 34: A sample hole triangulation.

Notice that for huge hole, the search of the optimal splitting plane may be time-consuming. For that reason, a speed up technique in huge holes is applied. A threshold aspect ratio is defined, which means that, we do not have to compute the aspect ratio of all the possible splitting planes. We only compute until we find one splitting plane with aspect ratio higher than the defined threshold.

In Figure 35 we show how it works out the filling-hole algorithm. On the left, one sample hole, and on the right, the generated triangular patch. We can notice that, the triangulation is not optimal. But, we choose speed in front of robustness. We consider also that, the triangulation is not optimal yet acceptable. From the user point of view is much better a uniform model without holes that if a small region is not optimally triangulated.

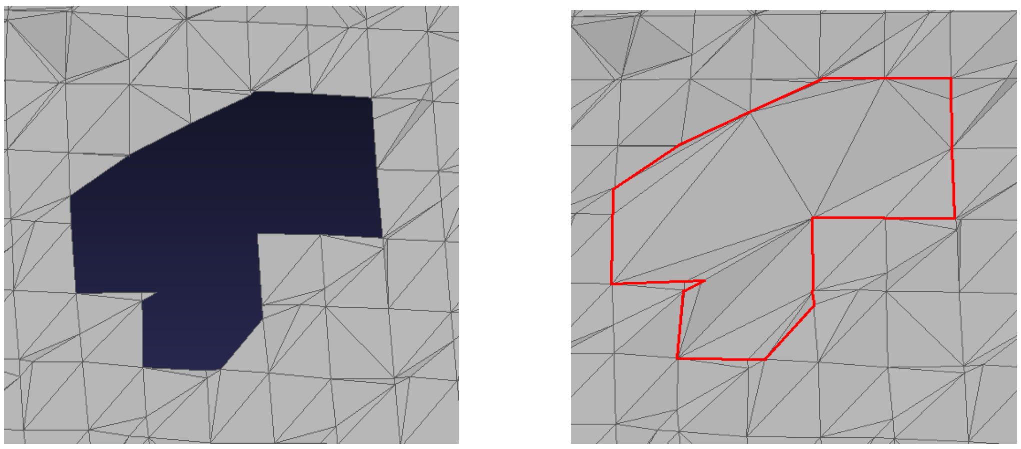


Figure 35: Left: A sample hole of a model reconstruction. **Right:** A succesful filling-hole triangulation.

In Figure 36 on left we show a bigger and more complex hole that the previous one. In this case, our hole-filling algorithm fails in the hole reconstruction. In Figure 36 on the right, we show that the algorithm is only able to partially triangulate the hole. By definition, we already know that not in all situations will be successfully triangulate.

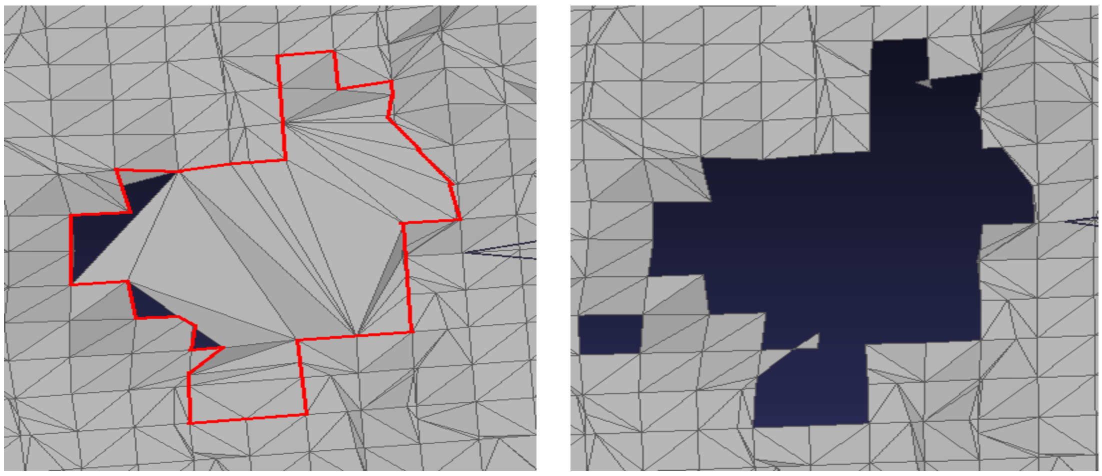


Figure 36: Left: A complex hole from a model reconstrunction. **Right:** The fail triangulation of hole-filling algorithm**.**

If we pay attention to the shape of the hole, we can notice that there are two triangles that are only connected to the mesh by one vertex. These triangles difficult pretty much the triangulation of the hole and are source of failures. For that, a small improving technique is implemented.

If the hole filling algorithm fails in the triangulation, we check in the hole boundary if exist some triangles with this characteristic (that is only connected by one if their vertices). If there are, we remove these triangles from the mesh, and from the hole loop. Then, we try to triangulate once again the cleaned hole. We observe that applying it, quite more holes are properly triangulated. We can see the process in Figure 37.

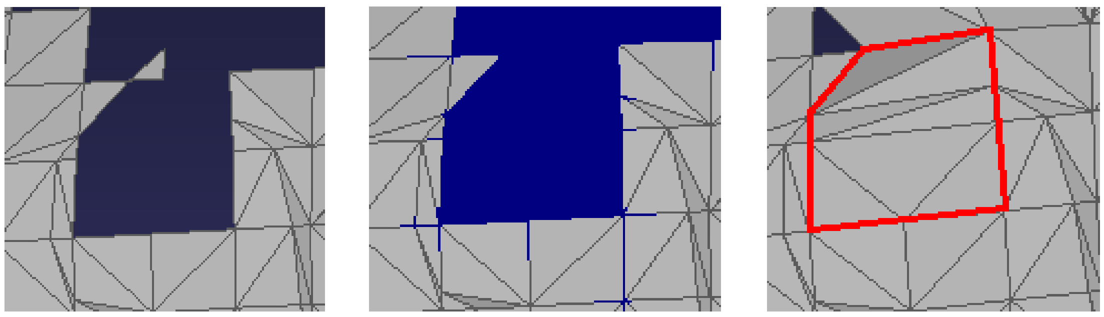


Figure 37: Original Hole. **Left: Original hole. Center:** Cleaned hole. **Right:** Hole triangulation.

4.4.4. Texture stitching

Many objects cannot be represented by a digital model that encodes just the shape characteristics. We also need to sample a texture map which gives to the model a more realistic appearance.

A texture is an image used to define the features of a surface an represents a RGB array of color values. The color values of a texture are referred to as *texels*. Texel's and pixels usually do not match. Sometimes, a pixel is the size of a fraction of a texel. In this case, the texture map needs to be stretched; this is called *magnification*. Sometimes, a pixel corresponds to multiple texels. In this case, the texture map needs to be shrunk; this is called *minification.*

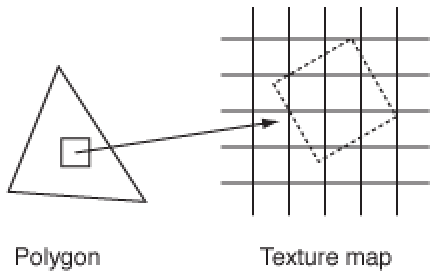


Figure 38: Texel minification

Typically, texture images are loaded from image files stored using a standard 2D image file format such as TIFF or JPEG.

Texture mapping [24] is a technique used for adding detail, surface texture or color to increase the realism of rendered objects. One source of textures are images of a real object, like the photos taken by a digital camera. The most of the objects may not be entire captured in a single image. Thus, as part of model reconstruction, we would like one technique that is able to combine multiple images of an object generating a single resulting one texture image.

The available color information is often managed by simply integrating the RGB images in a single texture, cutting and pasting parts of the photos in the geometry. The major problem of this approach, set aside the precision of mapping, is the color discontinuity in color intensity between different parts of the texture originated by the non uniform illumination conditions.

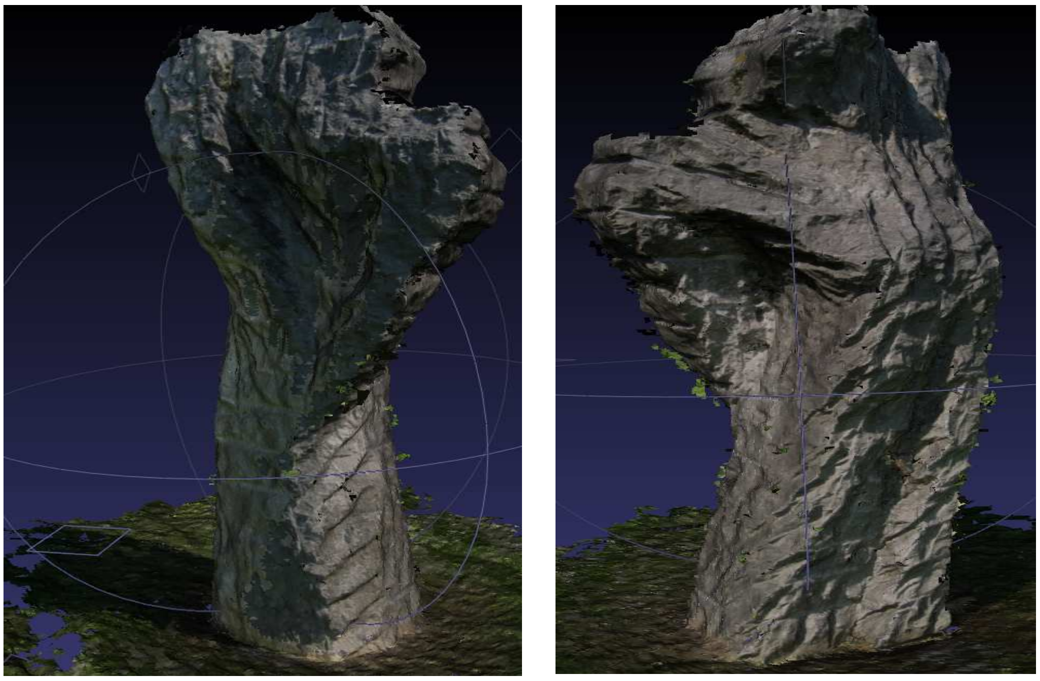


Figure 39. Texture color differences due to illumination conditions

The construction from a sequence of images of an optimal texture mapping on a 3D model is divided in two main steps [25]. The first is the image registration, where each image has to detect its position and orientation with respect to the real object. This phase is already done by the ARC Server [1] and used previously in other stages of the reconstruction procedure.

In the second phase, a texture image is reconstructed from the input images and mapped to the mesh. This texture should integrate the information available in the set of input images, possibly in a non-redundant manner and optimizing the sampled color representation. The aim is to reduce the possible texture distortion inherent in any image-to-surface mapping.

The input available data of the texture reconstruction process are the integrated triangle mesh and a number of images with the associated camera parameters. The texture stitching process is divided in three steps:

- Visibility calculation
- Patch generation
- Improving color matching and continuity

Visibility calculation is to find for every face of the 3D mesh the subset of cameras from which the face is visible, and the relative view angle of the camera.

Notice that, our intersection data structure used in the surface reconstruction step contains a field called mesh name (*nm)* where it is stored from which meshes this intersection originated. In the integrated mesh PLY file [30] we include a new property called *meshBit* where we store the value of the *nm* field of the intersection that actually, is the subset of originating mesh. At the same time, is the subset of cameras that the vertex is seen.

```
struct Intersection {
        float ic;
        int nm;
        bool sign;
        float dr;
        float weight;
        int meshPosition;
};
```

Furthermore, as we already know the camera parameters, the view angle between the camera position and the vertex, and the normal of the vertex can be easily computed. We use this information already previously. In summary, the first step of texture mapping already has all the required data without any extra implementation.

The second step is the patch generation, where the visibility information is used to subdivide the 3D mesh in sections. If we know the set of cameras that see a face, we can select the most orthogonal camera, since that is, the one with the most accurate color information (reducing the texture distortion). This selection of the most orthogonal camera view point for every face will divide the mesh in several patches. Each of these patches will be assigned to one input texture image. As a result, we can stitch the corresponding color information in the faces of the integrated mesh.

The last step, is improving color matching and continuity in order to obtain a better texture map, an intuitive objective is to reduce the number of mesh patches to avoid color differences. The goal is to have as large as possible continuous surface areas which are assigned to the same image. Notice that, it can than one camera that can provide a similarly orthogonal view with accurate definition of all the subset of valid cameras. That is another advantage of taking multiple overlapping images, it produces that several cameras seen the same parts of the object, so at this point, we can select the most suitable one according to our texture improvement interests.

This last step is not implemented in our system due to we did not have enough time to implemented it. Implementing the last step could one future work.

In Figure 40 we show an example where the texture stitching is applied to one 3D model. The model clearly has much more realistic appearance than without texture information. We would also like to comment that, small model reconstruction imperfections are less visible after applying color information.

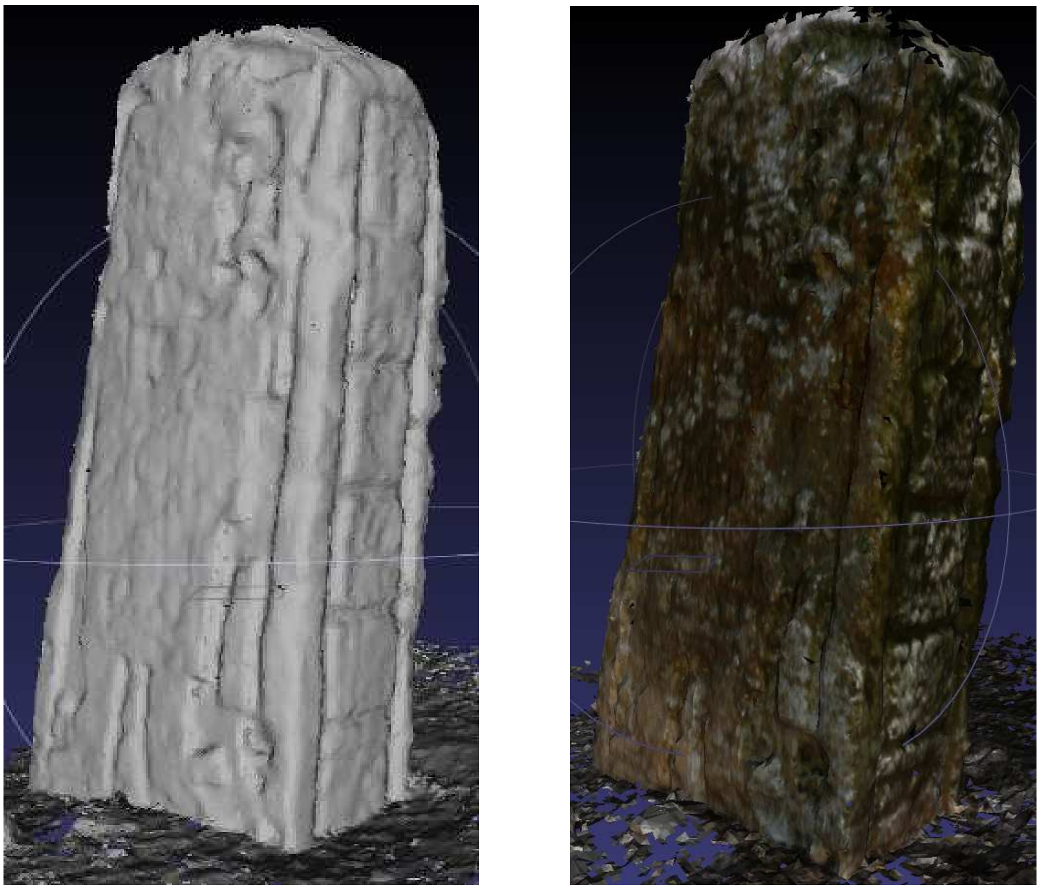


Figure 40: Left: A model reconstruction. **Right:** The model reconstruction after stitching texture color.

5. RESULTS

We proposed a volumetric integration technique based in the algorithm designed by Rocchini [4] called Marching Intersections. We observe that the resulting integrated model was not good enough, due to many holes and imperfections. For that reason, we suggested and presented some techniques to improve the model representation: Firstly, a clean-up and filter procedure of the input meshes. Secondly, a weighted intersections system related with the data acquisition confidence. After that, a hole-filling algorithm to repair the remaining holes in the integrated surface. Last, a texture stitching algorithm to give more realistic appearance to the model.

In this chapter, we analyze and show screenshots about how our integration algorithm works and behave. We use one sample object model available in the department called *Estela*. We begin from a reconstruction using only the original Marching Intersections algorithm, and adding progressively, the different proposed improving techniques.

Finally, we will compare our reconstruction results with VripPack volumetric algorithm and try to determine which the differences are.

5.1 Filtering of the input meshes

In this section we compare the improvements on the final reconstructed model if we integrate previously filtered meshes or if we integrate the same meshes without being filtered.

The input meshes usually have some floating parts, noisy and sharp features that difficult the reconstruction. We expect that, if we clean up the mesh from floating parts, noise and we smooth the sharp features, the final reconstructed model will visibly improve. We remember that, the mesh filter procedure is based in a cleanup phase and a bilateral filtering of the depth maps.

The clean-up step is used to remove floating parts and close small holes in the mesh, as to erode the mesh boundaries. In the other hand, the bilateral filtering is used to smooth the surface and remove noise in the mesh.

In Figure 41 on the left we have the *Estela* front-side reconstruction computing the Marching Intersections algorithm with the original meshes. On the right, we have the *Estela* front-side reconstruction computing the Marching Intersections algorithm with the filtered meshes.

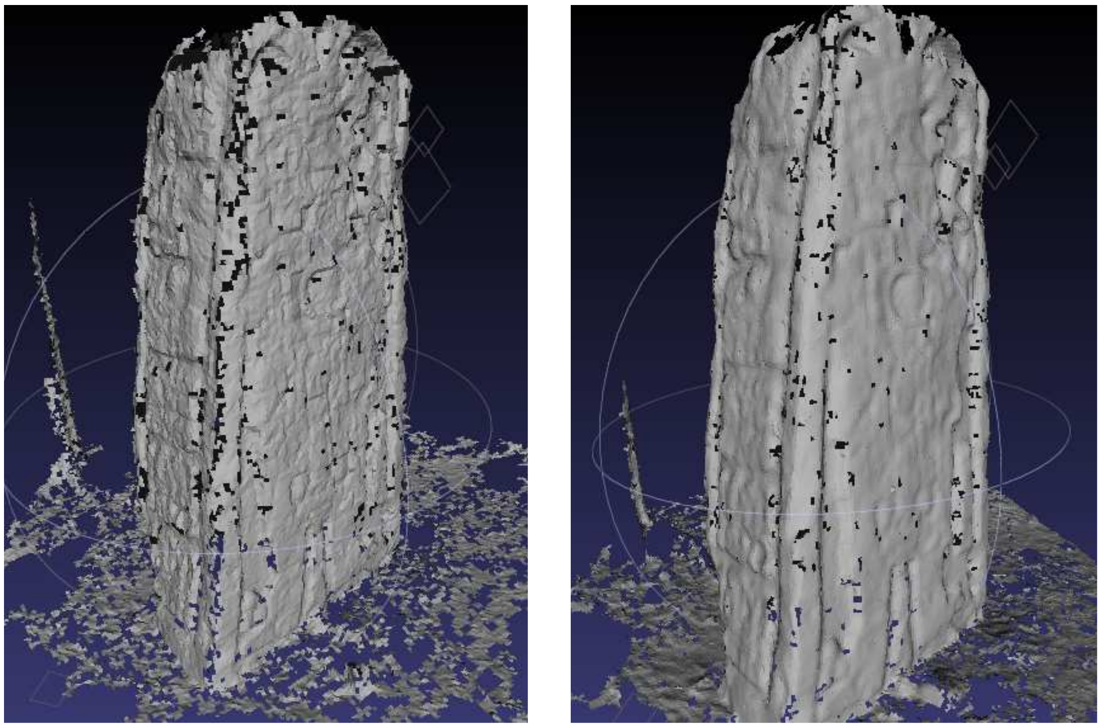


Figure 41: Left: Estela front-side reconstruction using original meshes. **Right:** Estela front-side reconstruction using filtered meshes.

We observe that the reconstruction model with the filtered meshes is better than with the original meshes. We justify this improvement for:

First, the mesh boundaries are generally, less confident areas. For that, in the cleanup process, we erode the mesh boundaries.

Second, the noise in the surface generates wrong intersections in the grid lines. This produces more cell failures. As a result, we have more holes in the model. For that, we use a bilateral filtering of the depth maps to remove the noise in the surface.

Third, computing blob detection in the input meshes, we achieve a more accurate model removing the floating parts of the input meshes. If not, some faces are over around the object due to the input floating parts.

As a summary, the filtering step provides a more accurate input meshes. A more accurate data directly implies more reliable intersections in the grid lines. Then, the probability to have cell failures decreases. As a result, our reconstruction has fewer holes. In the other hand, take in consideration that, due to the filtering, we lose some feature detail of the object. We consider this lose acceptable because the amount of smoothing applied is limited.

Finally, we can observe that our model reconstruction has improved. But still is not good enough because has too many holes.

5.2 Weighted intersections

In this section we compare the improvements in the final reconstructed model if we assign different weights to every intersection instead of a default value for all of the intersections.

Remember that, in the discretization step, we detect the intersections between the input meshes and our reference grid. Each intersection has a *weight* field that until now was set by default to one. Now we will use this field to assign different *weight* to each intersection according to the data acquisition confidence.

```
struct Intersection {
        float ic;
        int nm;
        bool sign;
        float dr;
        float weight;
        int meshPosition;
};
```

The parameters that we use to evaluate the confidence of one intersection is the quality count value (how many cameras is this point seen) and the view angle between the camera and the vertex normal (as more orthogonal, more confidence), and the final weight value is just the product of the two values. Additionally, we know that mesh boundaries are less reliable data. We expect that, the reconstruction will improve if we assign a minimum weight value to the intersections created in mesh boundaries.

If one cell fails, we assume that has some intersections originated in the merge process that result in a wrong cell configuration. This could be because some of these intersections became from meshes or areas less confident, like boundaries, less orthogonally camera angles or seen for few amounts of cameras.

In case of failing cell, we search in the cell the intersection the corner with the smallest weight, which is the least confident. After that, we compute the cell once again without taking in consideration the intersection less confident and check if now the configuration is correct. If it is correct, we look at the MC look up table and we add the patch surface to the integrated model.

If not, a second round is computed. We assume now that, the cell is wrong because after the merging one of the merged meshes adds some (note, more than one) intersections in the cell that produces the failure. So, as is more than one wrong intersection, we cannot detect it in the previous procedure. This time, we look in the cell for the mesh bit with the less confident intersection, and we compute once again the cell without considering the intersections that were originated from this mesh bit (we know that from the *nm* field). If now the cell is correct, we look at the MC look-up table and we add the patch surface to the integrated model.

If not, we leave it and go to the next cell. Then, we will have a non reconstructing cell, so a hole in the model.

We implemented these two techniques after the empirical observation about the cell failure reasons. We observe that, these two techniques are the ones that work better.

As a summary, we have three parameters to determine the confidence of one intersection: the camera view angle, the quality count value, and boundary or non-boundary intersection.

In Figure 42 on the top left, we have the *Estela* front-side model reconstruction without using the weighted intersections procedure. On the top center, we have the *Estela* front-side model reconstruction using the camera view angle as a confidence criterion. On the top right, we have the *Estela* front-side model reconstruction using the quality count value as a confidence criterion. On the bottom left, we have the *Estela* front-side model reconstruction using as a confidence criterion if is a boundary or non-boundary intersection. Finally, on bottom right, the *Estela* front-side model reconstruction after using the combining the three criteria's to set the intersection weight.

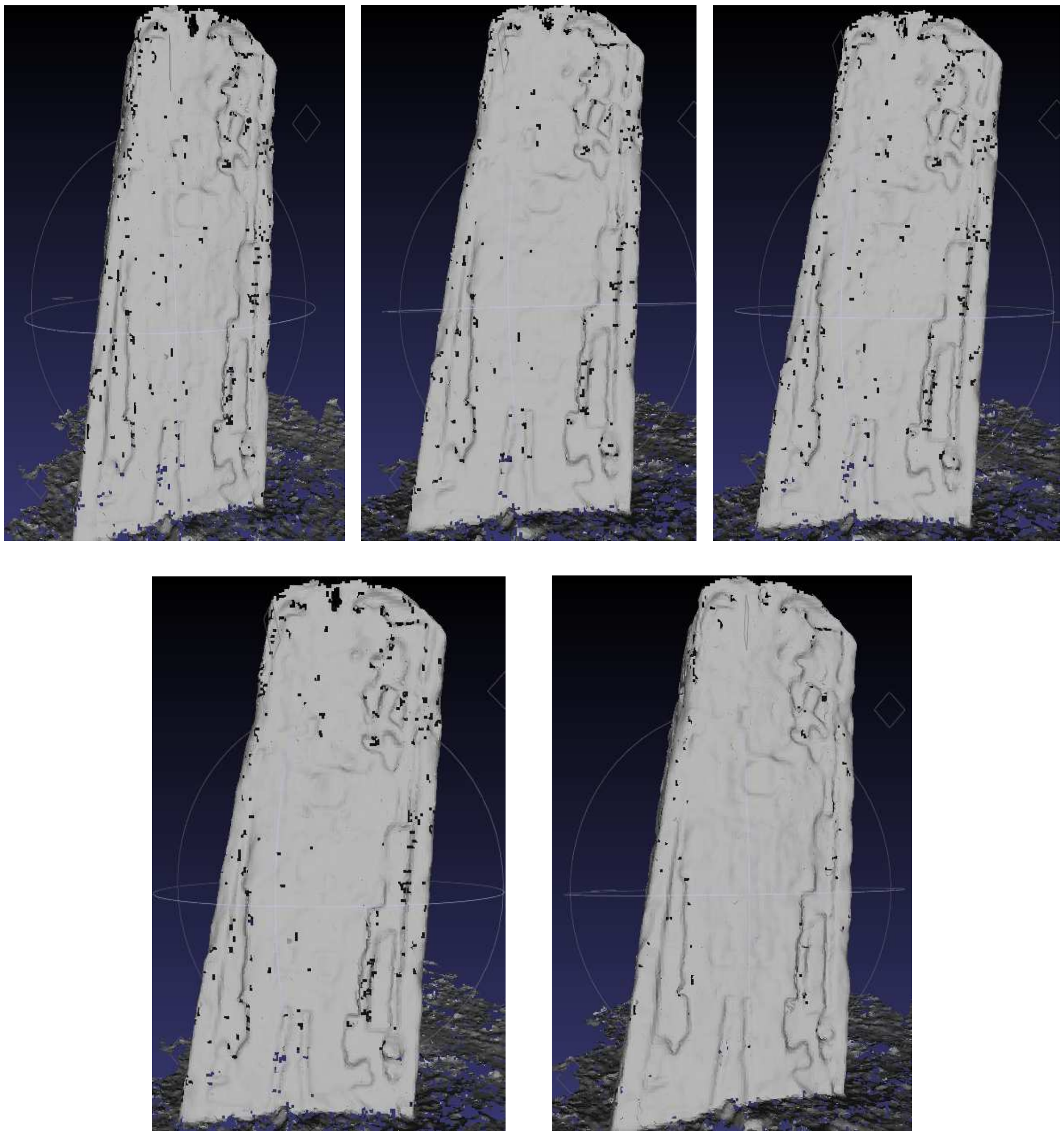


Figure 42: Top Left: Estela front-side model reconstruction using MI. **Top Center:** Estela front-side model reconstruction using only angle view weight criteria. **Top Right:** Estela front-side model reconstruction using only count weight criteria. **Bottom Left:** Estela front-side model reconstruction using only boundary weight criteria. **Bottom Right:** Estela front-side model reconstruction combining all the confidence criteria's.

We observe that the use of one of the weighting criteria separately (camera view angle, count map value or boundary set) in comparison to do not use weighting criteria is not a big difference. We notice that, there are some few less holes in the model reconstruction but not easily to appreciate.

Furthermore, we cannot determine which of the weighting criteria's work better separately because all of them give similar results. In the other hand, we observe that if we combine the three weighting criteria's, then the model reconstruction visibly improves. We can appreciate a reduction of holes in the model.

As a conclusion, we check empirically that, the use of weighted intersections only improves the final model if we combine together the three confidence criteria's. Our reconstruction is better, but still has some imperfections and holes.

5.3 Hole-filling algorithm

In this section we compare the reconstructed model according if we use or not a hole-filling technique.

The hole-filling algorithm is applied after the Marching Intersections finish and it is based in an iterative loop splitting hole approach characterized for being fast and efficient algorithm, but does not assure robustness.

We remember that, before compute the hole-filling algorithm, we need to locate the holes in the mesh. For that, during the execution of surface reconstruction, an auxiliary data structure records data related to each vertex. This data structure allows us to simply identify boundary vertices. Once a seed boundary vertex is identified, from it, a set of connected boundary edges can be traced until a close loop is generated, that is actually a hole.

We noticed that, using weighted intersections produces model reconstructions with fewer holes, and the remaining ones, are smaller and simpler than the previous ones. This lets us think that one mesh-based hole-filling algorithm will not affect our algorithm too much in terms of speed and efficiency. While, on the other hand, we expect to be able to close the most of the holes. As a result, we expect to achieve a big improvement to the final representation.

In Figure 43 on the left, we can see the *Estela* front-side model obtained with weighted intersections and on the right, the *Estela* front-side model after applying the hole-filling procedure:

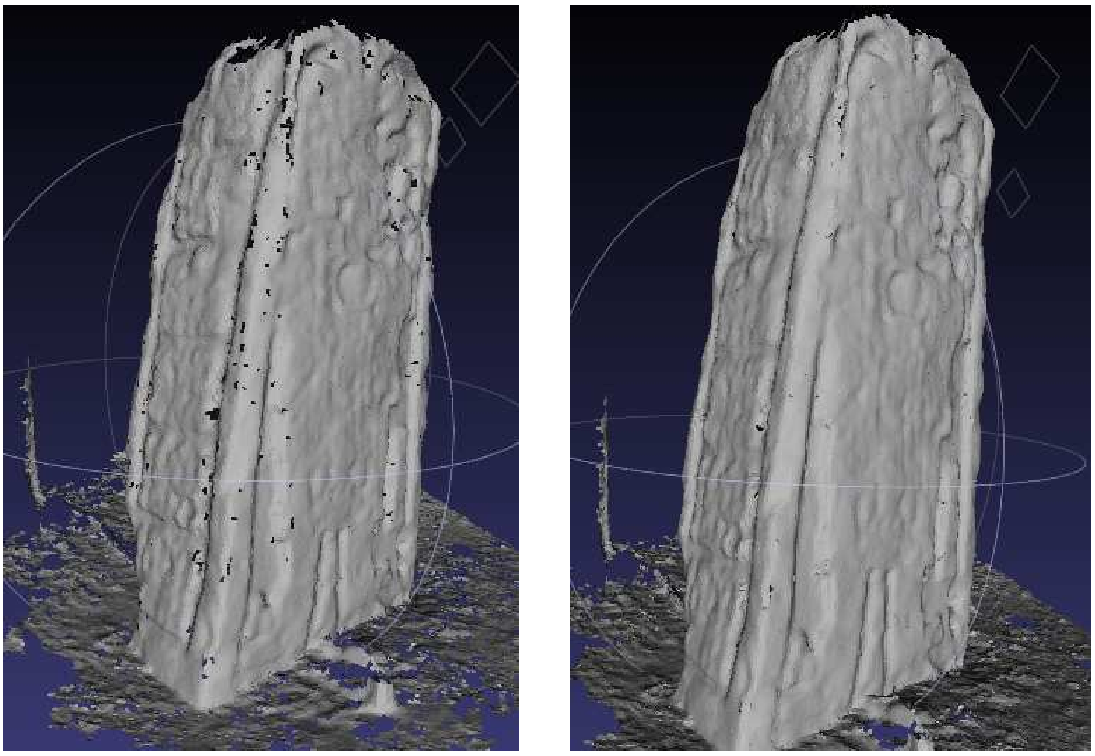


Figure 43: Left: Estela front-side model reconstruction without hole-filling. **Right:** Estela front-side model reconstruction with hole-filling algorithm

In Figure 44 on the left, we can see the *Estela* back-side model obtained with weighted intersections and on the right, the *Estela* back-side model after applying the hole-filling procedure:

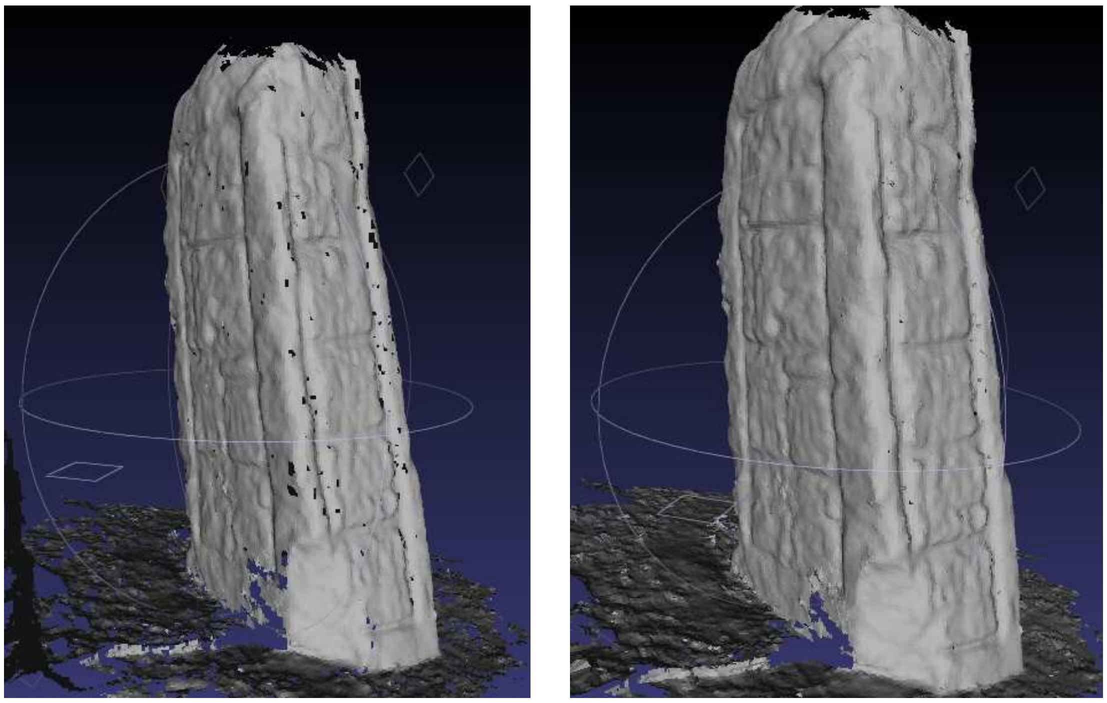


Figure 44: Left: Estela back-side model reconstruction without hole-filling. **Right:** Estela back-side model reconstruction with hole-filling algorithm

We observe that the hole-filling algorithm works pretty well. The algorithm is able to close almost all the holes of the mesh. The model reconstruction looks much better and the increment of computational cost due to the hole-filling algorithm is acceptable.

5.4 Texture Stitching

At this point, we want to provide more detail and real appearance to the model. For that, a texture stitching algorithm is implemented. It is based on computing a texture map from the original set of 2D images. According to the camera visibility areas, we divide the integrated mesh in patches, where each patch is assigned to one single texture image. In Figure 45, we have the model used until now after stitching texture color information. The object representation is much more realistic with texture information.

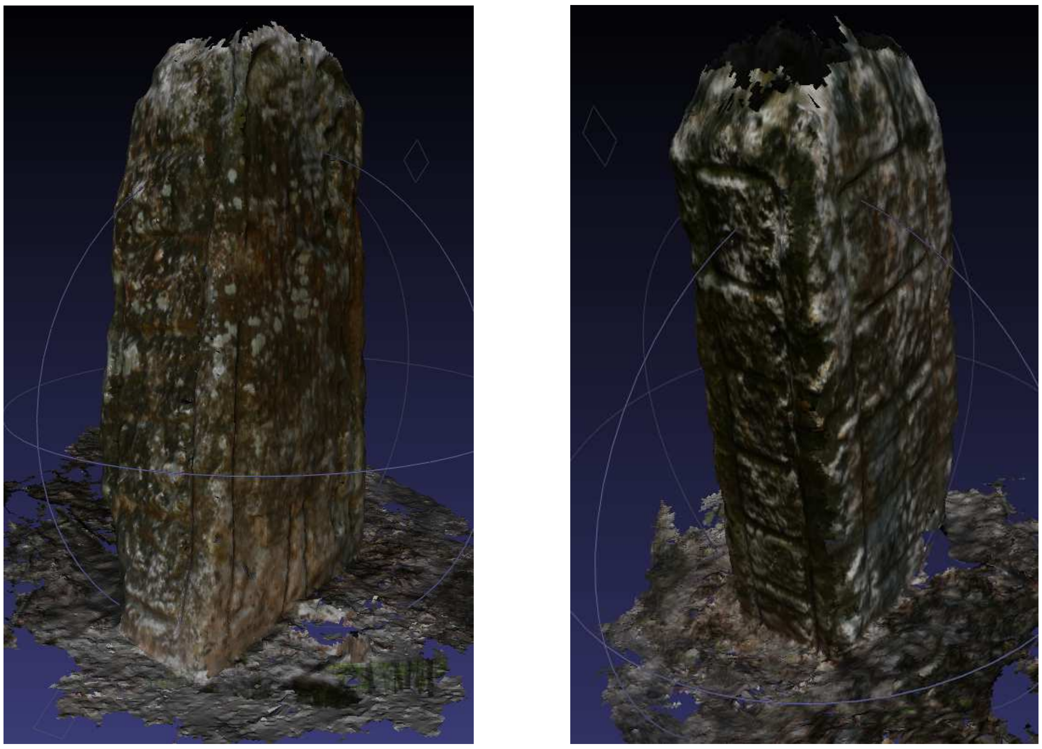


Figure 45: Left: Estela front-side model reconstruction adding texture detail**. Right:** Estela back-side model reconstruction adding texture detail**.**

5.5 VripPack comparison

It would be interesting to compare our reconstructions with the obtained reconstructions of other volumetric reconstruction algorithms. For that comparison, we chose Volumetric Range Image Processing Package (VripPack) that is a package for volumetrically merging a set of range images based on an implementation of the algorithm of Curless and Levoy [3]**.**

We talked about Curless and Levoy [3] previously in the integration methods overview. As a reminder, their volumetric representation consists of a cumulative weighted signed distance function. With this weight they to take into account the reliability of the data based on the characteristics of acquisition sensor. Working with one range image at a time, they first scan-convert it to a distance function, then combine this with the data already acquired using a simple additive scheme. They generate the final integrated mesh by extracting an isosurface from the volumetric grid. To fill gaps in the model, they tessellate over the boundaries between regions seen to be empty and regions never observed. Using this method, they were able to integrate a large number of range images in high-detail models.

The VripPack implementation is a set of source code, scripts, and binaries for creating surface reconstructions from range images. This package has the following features:

-Merges range images into a compressed volumetric grid.
-Extracts a surface from the compressed volumetric grid.
-Can fill holes in the reconstruction by carving out empty space.
-Removes small triangles from the reconstruction.

For the comparison, we compute a integration of the same *Estela* model, using the same input meshes and the same resolution output. The Figure 46 shows the results:

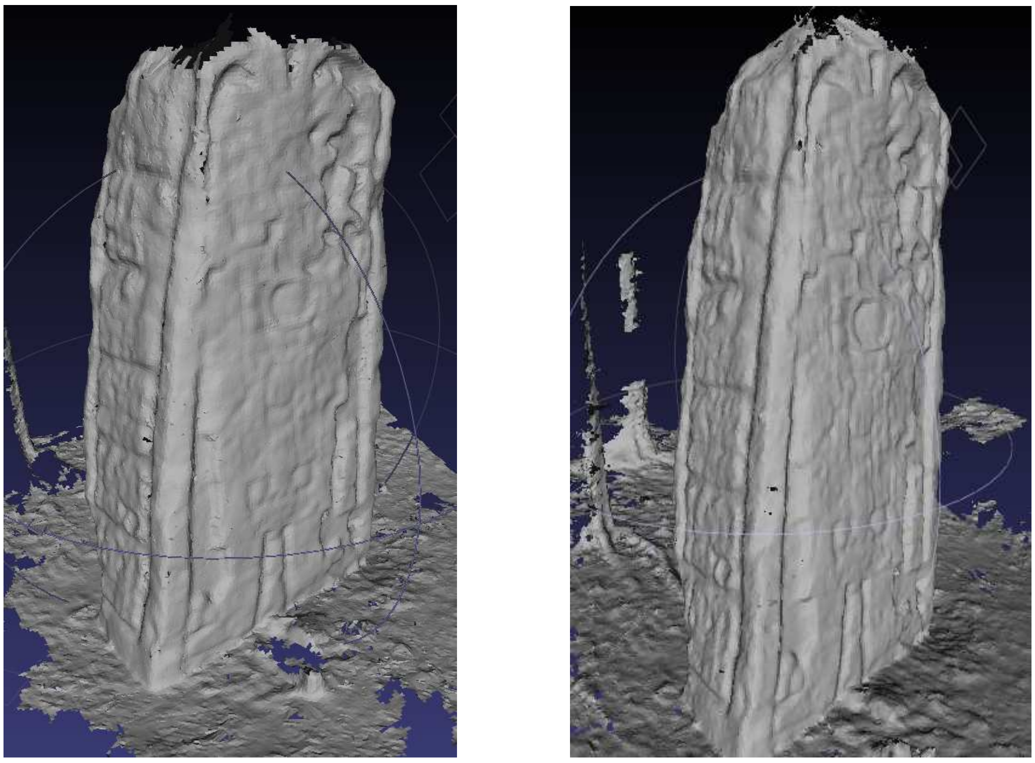


Figure 46: Left: Estela front-side model reconstruction using MI**. Right:** Estela front-side model reconstruction using VripPack.

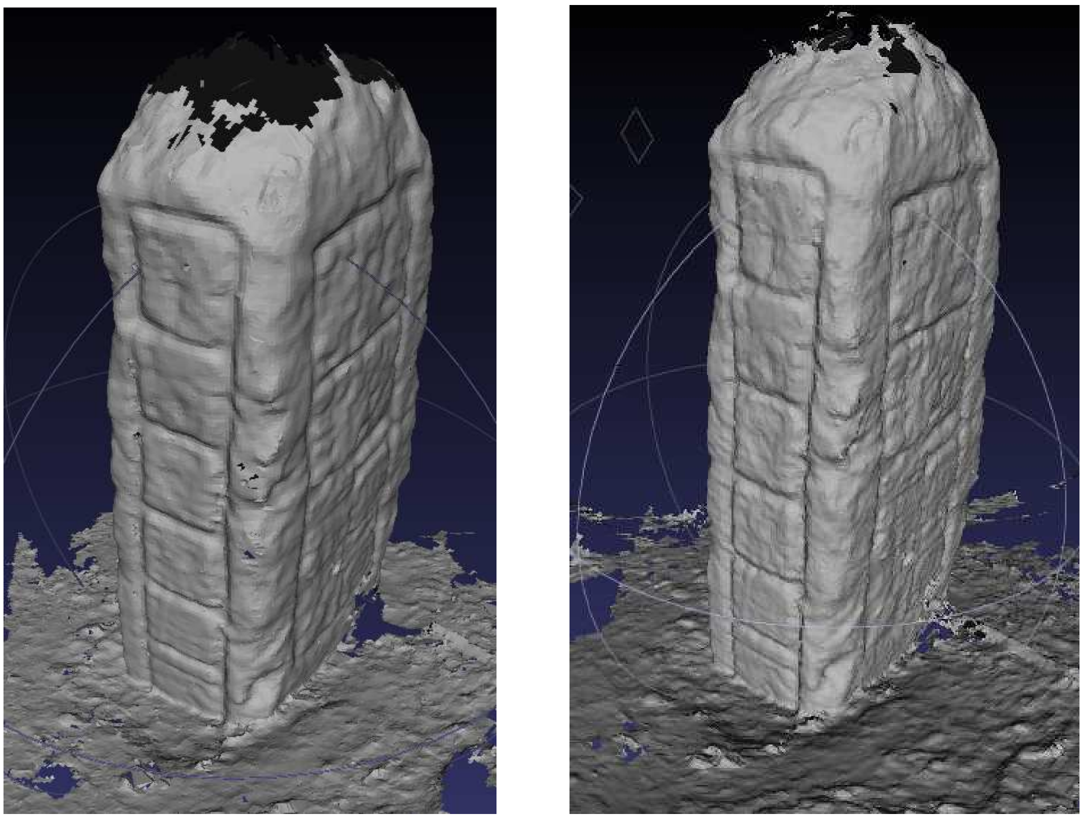


Figure 47: Left: Estela back-side model reconstruction using MI**. Right:** Estela back-side model reconstruction using VripPack.

After observing the results, we consider that our algorithm is perfectly comparable with VripPack. The reconstructed models do not have significant differences. To highlight something, could be that, VripPack obtains a bit less holes that our representation. In the other hand, VripPack presents some floating parts around the model surface. As is difficult to appreciate in Figure 47, in Figure 48 we enlarged one region with these floating parts.

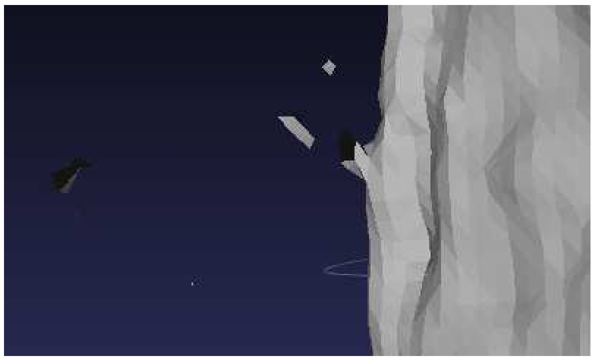


Figure 48: Left: Floating faces around VripPack Estela reconstruction.

We do a reconstruction of the Estela model reconstruction integrating eight meshes with a resolution of 300x300x300 (the reconstruction used in the screenshots).

The required time for the model reconstruction of the Estela using MI is the following:

```
MI Integration Time =     43 seconds.
```

The required time for filtering each input mesh before integrating them:

```
Mesh Filtering Time =     75 seconds.
```

Finally, the required time for adding texture color:

```
Texture Stitching Time =  59 seconds.
```

We do now, an Estela model reconstruction integrating the same eight meshes with VripPack algorithm:

```
VripPack Integration Time = 24 seconds.
```

Related to the resulting computation times, if we take into account only the integration process, VripPack is faster than Marching Intersections but still comparable. If we add the time for filtering the input meshes and texture stitching, our total computation time is much longer than VripPack's. That is because the previous filtering input mesh is not needed in VripPack and because the VripPack models do not include texture information.

We remark that, the Estela model has quite heavy depth maps. This produce that, the filtering step takes a significant time in the whole process.

In terms of memory consumption, our algorithm is less consuming as the integration procedure only records the intersections between the surface and the reference grid. While VripPack uses a volumetric voxel cumulative weighted signed distance function.

6. CONCLUSIONS

We have implemented a volumetric algorithm for the integration of multiple overlapping range images called Marching Intersections. The algorithm presents good characteristics in terms of efficiency, compactness, and low memory usage.

Marching Intersections locates intersections between the input meshes and the lines of the user selected grid, rather than in the distances of the input surfaces from the nodes of the grid. In the surface reconstruction step, the algorithm moves on the intersections and therefore only active cells are visited, without the need to store the addresses of the already visited cells. For these reasons, we consider the algorithm efficient.

Although the method adopts a volumetric approach, the discretized volume is not explicitly represented. Each intersection simply requires a floating point intercept (*ic*) and a bit for the sign (*sg*). For this reason, we consider the algorithm compact.

The memory required is proportional to the number of output vertices and does not depend on the complexity and the size of the input model. For this reason we consider the algorithm good in memory consumption.

Analogously to other voxel-based solutions, the size of the output mesh depends on the size of the 3D grid used in resampling and reconstruction. This is an advantage common to all voxel-based solutions because the user can produce representations at different resolutions, according to the selected grid. The resulting size of the fused mesh is in general much lower than the total size of the input range maps.

With respect to other voxel-based methods, we do not resample mesh geometry. This means that instead of performing a double conversion step (from mesh to voxel space, and again from voxel space to mesh), the geometry returned by Marching Intersections is obtained by the precise computation of intersections between input meshes and the lines of the grid.

On the other hand, we highlight the limits of the proposal. Marching Intersections algorithm is faster and more compact than other volumetric approaches, but it is certainly less robust. Its lower robustness mainly depends on the presence of noise in the input data rather than on the choice of maximum merge distance or the grid size. Maximum merge distance can be easily fixed from the sampling density and the residual registration error. Noisy data, generally border faces and sharp areas in the range maps with wrong orientation, generate non-MC-compliant cell configurations, and therefore many holes. For that reason, some improving techniques are proposed:

The first one is a mesh filtering step. The filtering step provides a more accurate input meshes. A more accurate data directly implies more reliable intersections in the grid lines. Then, the probability to have cell failures decreases. As a result, our reconstruction has fewer holes. In the other hand, take in consideration that, due to the filtering, we lose some feature detail of the object. We consider this lose acceptable because the amount of smoothing applied is limited. We observed that our model reconstruction has improved. But still is not good enough because has too many holes.

The second one is a weighted intersection technique. We have three parameters to determine the confidence of one intersection: the camera view angle, the quality count value, and boundary or non-boundary intersection. We observed that the use of one of the weighting criteria separately (camera view angle, count map value or boundary set) in comparison to do not use weighting criteria is not a big difference. In the other hand, the use of weighted intersections improves the final model if we combine together the three confidence criteria's. Our reconstruction is better, but still has some imperfections and holes

The third one is a hole-filling algorithm. The hole-filling algorithm is applied after the Marching Intersections finish and it is based in an iterative loop splitting hole approach characterized for being fast and efficient algorithm, but does not assure robustness. We observe that the hole-filling algorithm works pretty well. The algorithm is able to close almost all the holes of the mesh. After adding the weighted intersections, the remaining holes are smaller and simpler, so this algorithm is able to fill them properly. The model reconstruction looks much better and the increment of computational cost due to the hole-filling algorithm is acceptable.

We justify the remaining holes due to in surface reconstruction we use the simpler Marching Cubes look up tables, therefore, there are some ambiguity cells which surface patch could be not properly reconstructed, generating skinny triangles and weird shape holes.

The last improvement technique is adding texture color to the model to provide a more realistic representation. Furthermore we make a difference with VripPack integration that does not provide texture information. This procedure was implemented very efficiently due to we already know which cameras seen each vertex in the surface reconstruction in the *nm* intersection field. In the other hand, as is based in a patch stitching method, in the boundaries between patches could be color difference due to different illumination conditions in the images.

At this point, we compare our algorithm with VripPack volumetric algorithm, and we check that the resulting models reconstructions are comparable and pretty much the same. In terms of computation times, if we take into account only the integration process, VripPack is faster than Marching Intersections but still comparable. If we add the time for filtering the input meshes and texture stitching, our total computation time is much longer than VripPack's. But VripPack does not include texture color in the representation.

7. FUTURE WORK

In the surface reconstruction step, Marching Intersections uses the look-up tables from Marching Cubes algorithm in order to determine the patch surface inside each cell. In this work, we use the original Marching Cubes look-up tables. The original Marching Cubes look-up table presents some ambiguity situations where, the patch surface is not properly determined. A future work is to implement a new version of the look-up tables that solution of these ambiguity cases. The proper solving of these triangulation ambiguities will provide more accurate reconstructions.

The last improvement technique implemented was adding texture color information to the model to provide a more realistic representation. The major problem of our adding texture approach, set aside the precision of mapping, is the color discontinuity in color intensity between different parts of the texture originated by the non uniform illumination conditions. We did not have time implement an algorithm to solve color differences due to illumination conditions. Then, one future work is implementing an algorithm to improve color matching and continuity.

Our model reconstruction is delivered in PLY format. Other future work is to export the PLY model to other more popular formats, like Google Earth format. In this case, we make more easily available our model reconstruction.

8. REFERENCES

[1] Maarten Vergauwen and Luc Van Gool. *"Web-Based 3D Reconstruction Service*", Machine Vision Applications, 17, pages 411-426, 2006.

[2] J.-D. Boissonmat. *"Geometric structures for three-dimensional shape representation*", ACM Transactions on Graphics , 3(4):266-286, October 1984.

[3] B. Curless and M. Levoy. *"A Volumetric method for building compex models from range images*", In: Proceedings of SIGGRAPH'96 , New Orleans,4-9 August 1996, Comput Graph 30:303-312.

[4] C. Rocchini, P. Cignoni, F. Ganovelli, C. Montani, P. Pingi, R. Scopigno. *"The Marching Intersections algorithm for merging range images*", In: Technical Report B4-61-00 I.E.I – C.N.R., Pisa, Italy June 2000.

[5] H. Edelsbrunner and E.P. Mucke. *"Three-dimensional alpha shapes*", In: Workshop on Volume Visualization , pages 1802-1807, December 1987.

[6] Ratishauser M, Stricker M, Trobina M. *"Merging range images of arbritrary shaped objects*", In: Proceedings of the IEEE conference on computer vision and pattern recognition , Seattle, 20-24 June 1994, pages 573-580.

[7] Hoppe H, DeRose T, Duchamp T, McDonald J, Stuetzle W. *"Surface reconstruction from unorganized points*", In: Proceedings of SIGGRAPH'92, Chicago, 26-31 July 1992, Comput Graph 26(2): pages 71-78.

[8] C.L. Bajaj, F. Bernardini and G. Xu. *"Automatic reconstruction of surfaces and scalar fields from 3D scans*", In: Proceedings of SIGGRAPH'95, Los Angeles, CA, August 6 1995, ACM Press, pages 109-118.

[9] M. Soucy and D. Laurendeau. *"A general surface approach to the integration of a set of range views*", In: IEEE Transactions on Pattern analysis and Machine Intelligence, April 1995, 17(4): 344-358.

[10] G. Turk and M. Levoy. *"Zippered polygon meshes from range images*", In: Proceedings of SIGGRAPH'94, Orlando, July 24-29 1994 , ACM Press, pages 311-318.

[11] Pito R. *"Mesh integration based on co-measurements*", In: Proocedings of the international conference on image processing Laussane, Switzerland, 16-19 September 1996 , pages 397-400.

[12] Pulli K, Duchamp T, Hoppe H, McDonald J, Shapiro L, Stuetzle W. *"Robust meshes from multiple range maps*", In: Proocedings of the international conference on recent advances in 3D digital imaging and modeling, Otawa, Ontario, Canada, 12-15 May 1997. IEEE Press, New York, pages 205-211.

[13] C. I. Conolly . *"Cumulative generation of octree models from range data*", In: Proocedings of the international conference on Robotics, March 1984, pages 25-32.

[14] Hilton A, Soddart AJ, Illingworth J, Windeatt T. *"Multi-resolution geometric fusion*", In: Proocedings of the international conference on recent advances

in 3D digital imaging and modeling , Otawa, Ontario, Canada, 12-15 May 1997. IEEE Press, New York, pages 181-188.

[15] Wheeler MD, Sato Y, Ikeuchi K. *"Consensus surfaces for modeling 3D objects from multiple range images*", In: IEEE international conference on computer vision, Bombay, India January 1998.

[16] W. E. Lorensen and H.E. Cline. *"Marching Cubes: A high resolution 3D surface construction algorithm*", In: ACM Computer Graphics (SIGGRAPH'87 Proocedings) 1987, volume 21, pages 163-170.

[17] C. Tomasi, R. Manduchi. *"Bilateral Filtering for Gray and Color Images*", In: Proocedings of the IEEE International Conference on Computer Vision, Bombay, India 1998, pages 839-847.

[18] W. Zhao, S. Gao, H. Lin. *"A robust hole-filling algorithm for triangular mesh*", In: International Journal of Computer Graphics, China, 2007, pages 987-997.

[19] J. Davis, Stephen R. Marschner, Matt Garr, M. Levoy. *"Filling holes in complex surfaces using volumetric diffusion*", In: First International Symposium on 3D data processing, visualization and transmision. Padua, Italy, June 2002.

[20] Carr J, Beatson R., Cherrie J, Mitchell T, Fright W, McCallum B. *"Reconstruction and representation of 3D objects with radial basis functions*", In: Computer Graphics (Proc SIGGRAPH), 2001, pages 66-76.

[21] Liepa P. *"Filling holes in meshes*", In: Eurographics symposium on geometric processing ,pages 200-207.

[22] Jun Y. *"A piecewise hole filling algorithm in reverse engineering*", In: Comput Aided Des ,pages 263-270.

[23] William J. Schroeder, Jonathan A. Zarge, William E. Lorensen. *"Decimation of triangle meshes*", In: International Conference on Computer Graphics and Interactive Techniques, 1992, pages 65-70.

[24] C. Rocchini, C. Montani, P. Cignoni, R. Scopigno. *"Acquiring, Stitching and Blending Diffuse Appearance Attributes on 3D Models*", In: The Visual Compuper Journal, May 2002, pages 186-204.

[25] M. Callieri, P. Cignoni, R. Scopigno. *"Reconstructing textured meshes from multiple range rgb maps*", In: Visual Modeling and Visualization, November 2002, pages 419-426.

[26] M. Tarini, M. Callieri, C. Montani. *"Marching Intersections: An Efficient Approach to Shape-from-Silhouette*", In: Visual Modeling and Visualization, November 2002.

[27] Joshua Podolak and Szymon Rusinkiewicz .*"Atomic volumes for mesh completion*", In: Eurographics Symposium on Geometry Processing,2005.

[28] Chun-Yen Chen, Kuo-Young Cheng, Hong-Yuan Mark Liao. *"Fairing of polygon meshes via bayesian discriminant analysis*", In: International Conference in Central Europe on computer Graphics, Visualization and computer Vision,2004.

[29] Trolltech. *"Qt Reference Documentation (Open Soruce Edition)*", In: http://doc.trolltech.com/4.3/.

[30] Paul Bourke.*"Polygon File Format"*, In: http://local.wasp.uwa.edu.au/~pbourke/dataformats/ply/.